

Design and implementation of a modern algebraic manipulator for Celestial Mechanics

Francesco Biscani

January 30, 2008

“Design and implementation of a modern algebraic manipulator for Celestial Mechanics”, by Francesco Biscani (bluescarni@gmail.com), is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License (<http://creativecommons.org/licenses/by/3.0/>).



Copyright © 2007 by Francesco Biscani.

Created with L^AT_EX and X_YT_EX.

Dedicata alla memoria di Elisa.

Summary¹

The goals of this research are the design and implementation of a modern and efficient algebraic manipulator specialised for Celestial Mechanics. Specialised algebraic manipulators are fundamental tools both in classical Celestial Mechanics and in modern studies on the behaviour of dynamical systems, and they are routinely employed in such diverse tasks as the elaboration of theories of motion of celestial bodies, geodetical and terrestrial orientation studies, perturbation theories for artificial satellites and studies about the long-term evolution of the Solar System.

Specialised manipulators for Celestial Mechanics are usually concerned with mathematical objects known as Poisson series (see Danby et al. [1966]), which are defined as multivariate Fourier series with multivariate Laurent series as coefficients:

$$\sum_i P_i \begin{pmatrix} \cos \\ \sin \end{pmatrix} (i_1 y_1 + i_2 y_2 + \dots + i_n y_n),$$

where the P_i are multivariate polynomials. Poisson series manipulators have been developed continuously since the '60s and today there are many different packages available (an incomplete list includes Herget and Musen [1959], Broucke and Garthwaite [1969], Jefferys [1970, 1972], Rom [1970], Bourne and Horton [1971], Babaev et al. [1980], Dasenbrock [1982], Richardson [1989], Abad and San-Juan [1994], Ivanova [1996], Chapront [2003b,a] and Gastineau and Laskar [2005]). General manipulators, like Mathematica and Maple, are unsuitable for use in actual problems because their genericity implies a very high impact on performance (for many operations this slowdown can be estimated in three or four orders of magnitude). Many of the existing specialised manipulators, on the other hand, are much too focused on specific problems, and the general unavailability of the source code makes it hard to reuse and adapt existing manipulators for other problems. Besides, not always the data structures and the algorithms used in existing computer algebra systems are fast, leading to sub-optimal performance.

We propose here *Piranha*, a Poisson series manipulation framework designed to be extended and adapted for different purposes. It is written in C++ using template programming techniques (see Vandevoorde and Josuttis [2002]), it is object-oriented and uses paradigms like multiple inheritance, iterators, static polymorphism and operator overloading. *Piranha* is based on generic programming techniques, which means that it is possible to use arbitrary classes to represent the elements of Poisson series, as long as those classes behave in a predefined way (i.e., as long as they provide a defined set of methods). In generic programming language we can say that *Piranha* defines concept classes and that the actual implementation of such concepts are models (see Gregor et al. [2006]).

Piranha uses modern data structures for the representation of Poisson series. In particular hashing techniques are widely employed. The main hash function used in *Piranha* was conceived originally to operate on strings in databases (see Ramakrishna and Zobel [1997]), and has proved to be very effective for the representation of Poisson series. *Piranha* employs also the `multi_index_container` class from the Boost C++ libraries, which provides a flexible and efficient container class to store and order the terms of a series in multiple ways. Many other facilities provided by the Boost libraries are used.

¹This section is intended just as a condensed summary, for a more thorough introduction please see Chapter 1.

Piranha introduces also a new methodology for the computation of cosines and sines of Poisson series, based on an expansion into Bessel functions of the first kind known as Jacobi-Anger development (see Brown and Churchill [1993]). This kind of development allows to compute cosines and sines of a wide class of Poisson series, whereas the Taylor expansions commonly employed for this task in other manipulators are effective only for specific Poisson series.

Another original contribution, to the best of our knowledge, is the approach devised for the multiplication of Poisson series, which is based on a technique derived from the Kronecker algorithm and that we have called *coded arithmetics*, and which allows to speed up considerably the manipulation of trigonometric multipliers during multiplication.

We then introduce Pyranha, a set of bindings to use Piranha from the Python programming language (see Python). With Pyranha it is possible to leverage Piranha's capabilities from a comfortable and easy to use interpreted language. This way, by means of enhanced Python interfaces (like IPython), it is also possible to use Piranha interactively à la Mathematica or Maple, without the need to code in C++ but retaining the speed of a compiled language.

We also present some examples of use cases for a Poisson series manipulator. We show how Piranha can be used for the transformation of theories of motion and for the computation of the harmonic development of the tide-generating potential.

Finally, we briefly discuss Piranha's performance by comparing it to the well-known manipulators TRIP and PARI. The benchmarks are encouraging and show that Piranha is on the right track performance-wise. Optimizations and improvements for Piranha are then discussed, with particular focus on cache memory optimizations and memory allocation, in light of the fact that presently Piranha employs data structure implementations and memory allocators available in the C++ standard library (which are hence not optimized for the specific tasks of the manipulator). Beside performance improvements, also functionality extensions are also discussed, such as the implementation of still missing manipulation capabilities.

Riassunto

Introduzione

Gli scopi di questa ricerca sono il design e l'implementazione di un manipolatore algebrico specializzato per la Meccanica Celeste moderno ed efficiente. I manipolatori algebrici specializzati sono strumenti fondamentali sia nella Meccanica Celeste classica che nei moderni studi sui sistemi dinamici, e sono impiegati abitualmente in diversi campi: teorie del moto di corpi celesti, studi geodetici e di orientazione planetaria, teorie perturbative per satelliti artificiali e studi riguardanti l'evoluzione a lungo termine del Sistema Solare.

Serie di Poisson

I manipolatori algebrici specifici per la Meccanica Celeste solitamente si occupano di oggetti matematici noti come serie di Poisson (Danby et al. [1966]).

Le serie di Poisson sono definite come serie di Fourier multivariate con serie di Laurent multivariate come coefficienti:

$$\sum_i P_i(x_1, x_2, \dots, x_m) \begin{pmatrix} \cos \\ \sin \end{pmatrix} (i_1 y_1 + i_2 y_2 + \dots + i_n y_n),$$

dove le $P_i(x_1, x_2, \dots, x_m)$ sono polinomi multivariati a coefficienti complessi. In Meccanica Celeste le serie di Poisson con coefficienti puramente numerici sono note anche come serie di Fourier:

$$\sum_i C_i \begin{pmatrix} \cos \\ \sin \end{pmatrix} (i_1 y_1 + i_2 y_2 + \dots + i_n y_n).$$

Le serie di Poisson formano un gruppo abeliano sotto le operazioni di addizione e sottrazione, ma non sotto l'operazione di moltiplicazione. Non è infatti possibile, in generale, calcolare in forma finita l'operazione di inversione di una serie di Poisson. La moltiplicazione di serie di Poisson è svolta con l'ausilio delle elementari formule trigonometriche di Werner:

$$\begin{aligned} A \cos \alpha \cdot B \cos \beta &= \frac{AB}{2} \cos(\alpha - \beta) + \frac{AB}{2} \cos(\alpha + \beta), \\ A \cos \alpha \cdot B \sin \beta &= \frac{AB}{2} \sin(\alpha + \beta) - \frac{AB}{2} \sin(\alpha - \beta), \\ A \sin \alpha \cdot B \cos \beta &= \frac{AB}{2} \sin(\alpha - \beta) + \frac{AB}{2} \sin(\alpha + \beta), \\ A \sin \alpha \cdot B \sin \beta &= \frac{AB}{2} \cos(\alpha - \beta) - \frac{AB}{2} \cos(\alpha + \beta). \end{aligned}$$

Le serie di Poisson sono inoltre caratterizzate da una forma canonica che assicura la rappresentabilità in maniera univoca con il numero minimo di termini.

Le serie di Poisson in Meccanica Celeste solitamente appaiono negli ambiti delle teorie perturbative, dove la loro introduzione permette di applicare i metodi standard della dinamica perturbativa, quali ad esempio il metodo di averaging. Tipicamente la forma di serie di Poisson è

mantenuta in tutte le fasi dell'elaborazione di una teoria del moto, e anche le soluzioni finali sono espresse in forma di serie di Poisson o serie di Fourier.

L'operazione di gran lunga più dispendiosa nella manipolazione di serie di Poisson, sia in termini di potenza di calcolo che di utilizzo della memoria, è la moltiplicazione. La complessità di questa operazione è quadratica, $O(n^2)$ (mentre addizione e sottrazione hanno complessità lineare, $O(n)$), e pertanto si rende necessaria l'adozione di metodologie di troncamento delle serie per evitare la crescita esplosiva del numero di termini durante le moltiplicazioni.

Operazioni avanzate sulle serie di Poisson

In Meccanica Celeste tipicamente sono necessarie manipolazioni sulle serie di Poisson più complicate rispetto alle operazioni fondamentali di somma e moltiplicazione. In particolare è necessario poter calcolare l'inversione e la radice quadrata di serie di Poisson. Queste due operazioni possono essere affrontate in maniera unificata ricorrendo al teorema binomiale generalizzato di Newton (Arfken and Weber [2005]), che permette di ricondurre l'elevamento di una serie di Poisson ad una potenza reale ad uno sviluppo in potenze naturali (e quindi a moltiplicazioni). Questo approccio è appropriato in Meccanica Celeste perchè le serie soggette ad elevamento a potenza reale rappresentano tipicamente le distanze dei corpi celesti, le quali, nella maggioranza dei casi, sono costituite da un termine costante dominante (la distanza media) e da una coda perturbativa.

Un'operazione utile nel contesto della manipolazione di teorie del moto esistenti è il calcolo del coseno (o seno) di una serie di Poisson. Per effettuare questo calcolo proponiamo una procedura che, per quanto abbiamo avuto modo di verificare, non è stata utilizzata in altri manipolatori. Questa procedura è basata sullo sviluppo di Jacobi-Anger (Brown and Churchill [1993], Weisstein [2007]), che permette di esprimere coseni e seni di coseni e seni come serie di Poisson in cui i coefficienti sono funzioni di Bessel del primo tipo. Il vantaggio di questo approccio risiede nel fatto che è molto più generale rispetto alla metodologia comunemente adottata (sviluppo in serie di Taylor) che è applicabile solo a serie di Poisson con caratteristiche peculiari.

Le operazioni trigonometriche e l'elevamento a potenza reale permettono poi di calcolare funzioni speciali di serie di Poisson, quali funzioni associate di Legendre e armoniche sferiche, utili nel contesto della Meccanica Celeste. L'implementazione del teorema di rotazione delle armoniche sferiche (Wigner [1931]), utile nel contesto della trasformazione di teorie del moto, è discussa brevemente.

Design di un moderno manipolatore di serie di Poisson

Lo sviluppo di manipolatori di serie di Poisson comincia alla fine degli anni '50, e nel corso degli anni vari pacchetti software sono stati sviluppati (una lista incompleta include Herget and Musen [1959], Broucke and Garthwaite [1969], Jefferys [1970, 1972], Rom [1970], Bourne and Horton [1971], Babaev et al. [1980], Dasenbrock [1982], Richardson [1989], Abad and San-Juan [1994], Ivanova [1996], Chapront [2003b,a] e Gastineau and Laskar [2005]).

I manipolatori generici, come ad esempio Maple e Mathematica, sono inadatti per l'utilizzo nell'ambito della Meccanica Celeste perchè la loro genericità implica una consistente riduzione

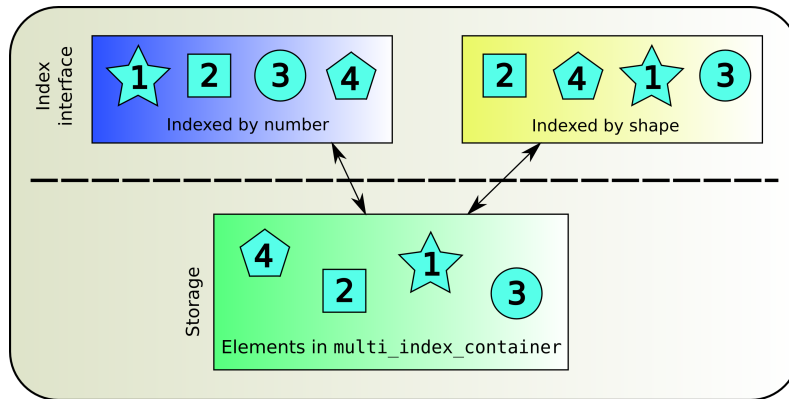


Figure 0.1: Schema di un `boost::multi_index_container` in cui sono definiti due ordinamenti differenti.

delle prestazioni, stimata in 3-4 ordini di grandezza. I software specializzati esistenti, d'altra parte, sono spesso troppo specifici per la risoluzione di un determinato problema, e la generale impossibilità di ottenere il codice sorgente rende impossibile il riutilizzo e l'adattamento di soluzioni esistenti. I linguaggi di programmazione normalmente impiegati, inoltre, sono di carattere procedurale e rendono difficoltoso il riutilizzo del codice disponibile. Molti dei manipolatori esistenti, infine, usano strutture dati e algoritmi non ottimali.

Questi sono i motivi che hanno condotto all'ideazione di *Piranha*, un framework di manipolazione per serie di Poisson. *Piranha* è scritto in C++ usando tecniche di programmazione generica tramite template (Vandevoorde and Josuttis [2002]) e programmazione a oggetti. Le librerie C++ *Boost* (Boost) sono impiegate estensivamente.

Per quanto concerne algoritmi e strutture dati, *Piranha* fa un uso estensivo di tecniche di hashing per l'identificazione dei termini, mentre per l'ordinamento delle serie vengono impiegati alberi a ricerca binaria auto-bilanciati (*self-balancing binary search trees*). L'ordinamento delle serie è necessario per l'implementazione efficiente di metodologie di troncamento da applicare durante le moltiplicazioni. La struttura base di una serie di Poisson in *Piranha* è costituita da una classe contenitore disponibile nelle librerie Boost, chiamata `multi_index_container`, che permette di definire semantiche di accesso e ordinamenti multipli per gli elementi contenuti (vedi Figura 0.1).

Piranha: architettura e dettagli di implementazione

Piranha è basato sulla programmazione generica tramite template. Ciò significa che gli elementi della architettura del manipolatore (i termini delle serie, i coefficienti, le parti trigonometriche, etc.) non sono prefissati: qualsiasi classe può essere utilizzata come elemento dell'architettura, a patto che soddisfi certi requisiti. In gergo, si dice che l'architettura di *Piranha* definisce dei *concetti* (Gregor et al. [2006]), mentre le implementazioni di tali concetti costituiscono un *modello*. In C++ un concetto è definito implicitamente dalla richiesta dell'implementazione di un determinato set di metodi e membri: se una classe non implementa un concetto, questa classe non può

essere usata in altre classi e routine generiche che richiedono l'implementazione di tale concetto. Il controllo sull'implementazione di un concetto in C++ avviene al momento della compilazione (o, in altri termini, i concetti sono *statici*). Ciò permette al compilatore di effettuare efficaci procedure di ottimizzazione ed evita qualsiasi tipo di overhead al runtime.

Attualmente l'architettura di Piranha definisce i concetti di:

- coefficiente,
- parte trigonometrica,
- termine.

Il concetto di coefficiente, ad esempio, richiede le operazioni di moltiplicazione per un altro coefficiente e di divisione per un numero intero, al fine di implementare le formule trigonometriche di Werner. Le parti trigonometriche, invece, devono implementare il concetto di somma e differenza trigonometrica.

L'architettura di Piranha prevede la presenza di una classe base che contiene i termini delle serie di Poisson. Questa classe viene ereditata da una classe specializzata che eredita in maniera multipla da altre classi, chiamate *toolbox*, le quali forniscono le implementazioni di funzioni avanzate e specializzate per determinati tipi di serie (vedi Figura 0.2). La classe base, infatti, implementa solo le operazioni fondamentali sulle serie di Poisson (I/O, inserimento termini, etc.). In un certo senso questo modello può essere visto come la messa in pratica di un'idea espressa in Henrard [1988] (per quanto abbiamo avuto modo di verificare, nessun altro manipolatore implementa un'architettura a livelli di questo tipo).

La funzione di hash adottata in Piranha è stata ideata per l'uso nei database e proposta in Ramakrishna and Zobel [1997], e si è rivelata estremamente performante per l'uso nelle serie di Poisson. Per aumentare le performance Piranha fa uso inoltre di istruzioni vettoriali, in particolare delle istruzioni SSE2 dei moderni processori Intel.

Manipolazione di polinomi sparsi multivariati

Un altro contributo presentato in questo lavoro è l'implementazione di una procedura efficiente per la moltiplicazione di polinomi multivariati sparsi e serie di Poisson, basata sull'algoritmo di Kronecker. L'idea è quella di codificare i monomi (o le parti trigonometriche delle serie di Poisson) tramite il seguente schema, chiamato *ordinamento m -variato n -lessicografico* (questo esempio si riferisce al caso $n = m = 3$):

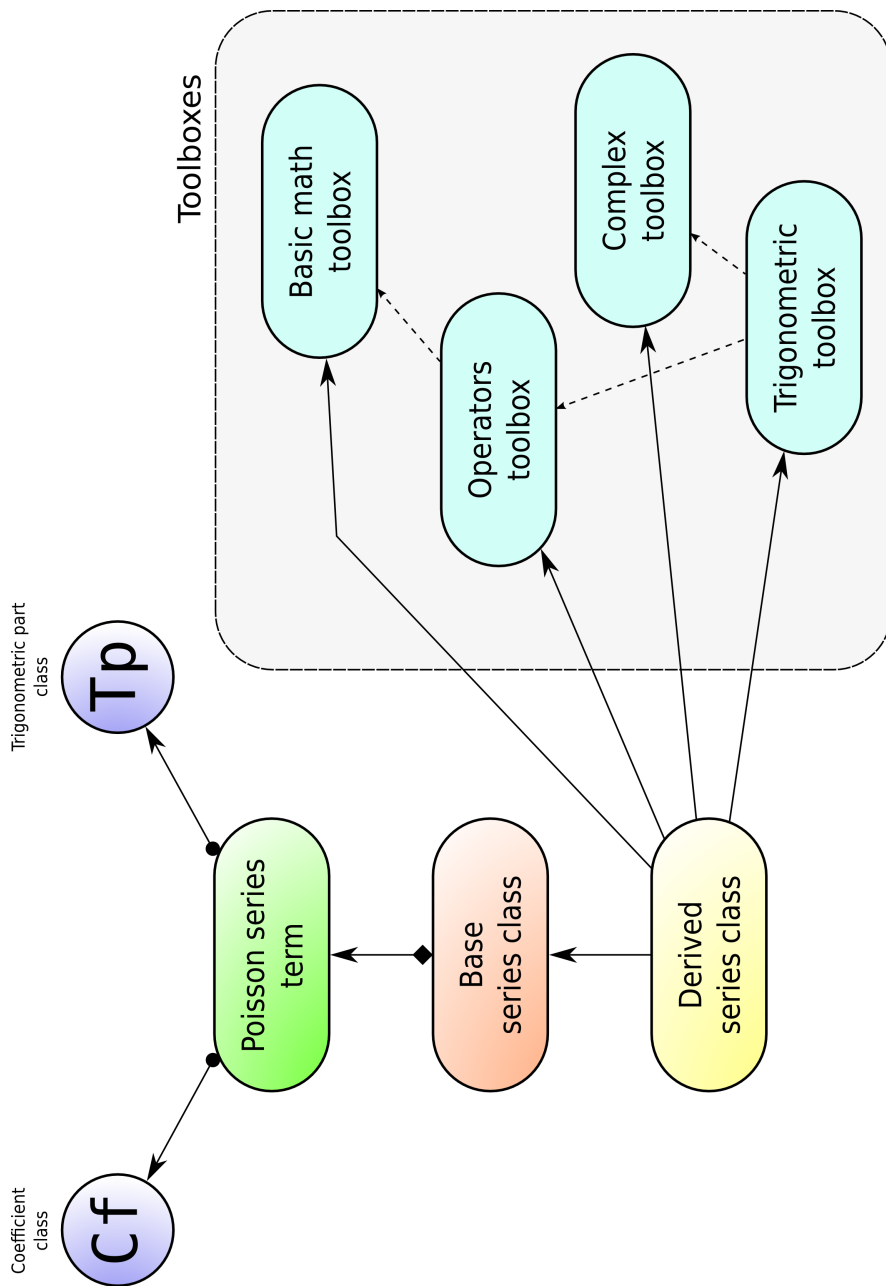


Figure 0.2: Gerarchia delle classi nell'architettura di Piranha.

x	y	z	Codice
0	0	0	0
0	0	1	1
0	0	2	2
0	0	3	3
0	1	0	4
0	1	1	5
0	1	2	6
0	1	3	7
0	2	0	8
0	2	1	9
0	2	2	10
0	2	3	11
...
3	3	3	63

Si dimostra che, sotto certe condizioni, si può effettuare l'aritmetica degli esponenti direttamente sui codici, abbassando quindi la complessità dell'operazione da $O(m)$ a $O(1)$. Inoltre il codice costituisce anche una funzione di hash perfetta (nel senso che c'è una relazione biunivoca fra ciascun set di esponenti e il rispettivo codice), e quindi la rappresentazione codificata si presta bene per l'utilizzo in tabelle hash.

Pyranha: Piranha dentro Python

Per facilitare l'utilizzo di Piranha, abbiamo scritto uno strato software che permette di accedere al manipolatore direttamente dal linguaggio Python (Python). Con Pyranha è possibile utilizzare Piranha da un linguaggio interpretato di livello più alto rispetto al C++, senza però sacrificare le performance di un linguaggio compilato.

È possibile utilizzare Pyranha da interfacce interattive Python avanzate, come IPython, che, in congiunzione con librerie grafiche come matplotlib e di librerie GUI come PyQt, permettono di lavorare in un ambiente grafico interattivo simile a quelli offerti da prodotti come Maple e Mathematica (vedi Figura 0.3).

Applicazioni

In questo capitolo vengono discusse due applicazioni di Piranha a problemi di Meccanica Celeste.

La prima applicazione consiste nel calcolo dello sviluppo armonico del potenziale generatore di marea (TGP) nel sistema Sole-Terra-Luna. La procedura è totalmente analitica e basata sulla teoria lunare ELP2000, e si presta ad estensioni e miglioramenti tramite l'adozione di un modello fisico più accurato che includa effetti di nutazione, perturbazioni planetarie e di figura, etc.

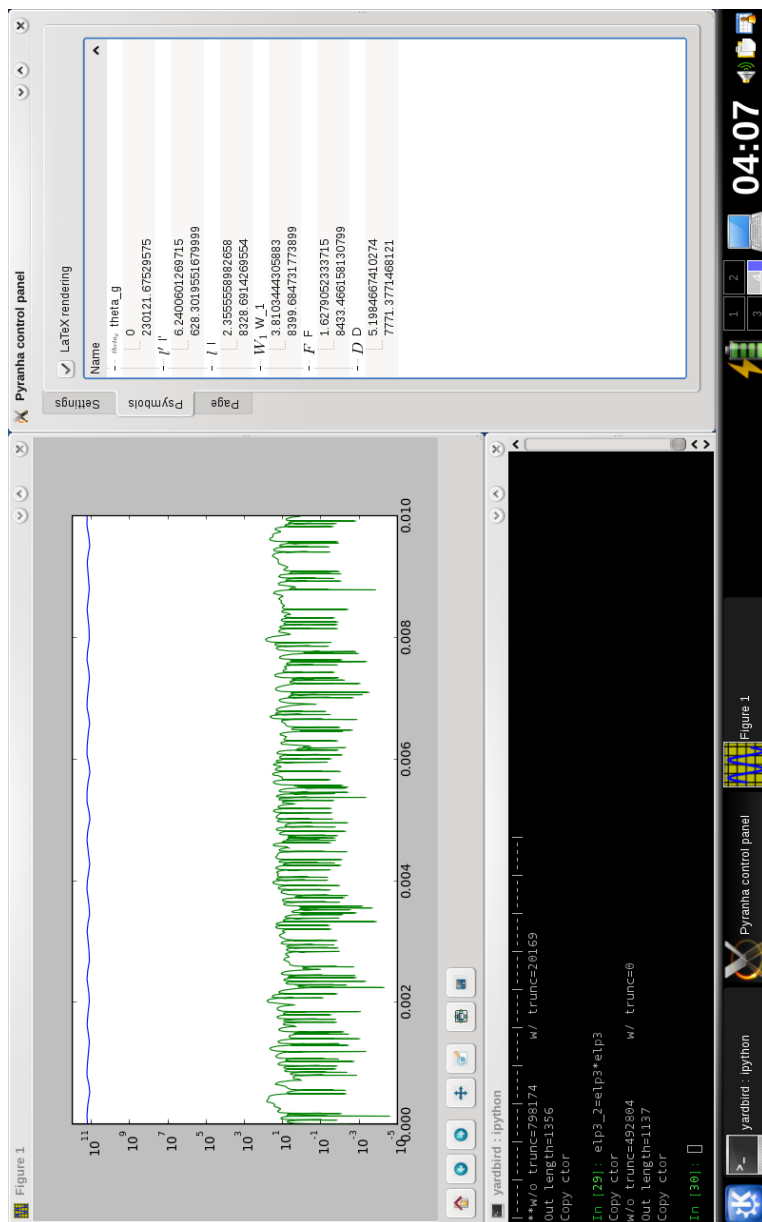


Figure 0.3: Uno screenshot di Pyranha utilizzato con IPython, matplotlib e PyQt in ambiente GNU/Linux.

La seconda applicazione è la trasformazione della teoria TASS (satelliti del sistema di Saturno) in una forma adatta per l'utilizzo in una teoria analitica del moto di un satellite artificiale o di una piccola luna. La complicata sequenza di operazioni da effettuare per ottenere la forma desiderata è descritta, ed i limiti della teoria TASS in questo contesto sono menzionati.

Prestazioni e sviluppi futuri

Le prestazioni di Piranha sono comparate a quelle di software maturi, come il manipolatore per la Meccanica Celeste TRIP e il manipolatore polinomiale Pari/GP. I benchmark indicano che le prestazioni di Piranha sono comparabili e, in certi casi, migliori. Ricordiamo che Piranha è in sviluppo da relativamente poco tempo, e una parte minima dello sviluppo è stata finora dedicata all'ottimizzazione delle performance.

Gli sviluppi futuri di Piranha verteranno sull'ottimizzazione (in particolare verso un utilizzo efficiente della memoria cache, verso l'adozione di strutture dati allo stato dell'arte, come il *cuckoo hashing*, verso la parallelizzazione del codice e verso un utilizzo più pervasivo delle istruzioni vettoriali dei moderni processori), e sull'implementazione di funzionalità di più alto livello ancora mancanti.

Viene infine discussa la possibilità di collaborazione con altri progetti open-source di manipolazione algebrica, in particolare con il progetto SAGE.

CONTENTS

1	Introduction	1
1.1	Motivation	1
1.2	Piranha	2
1.3	Structure of the dissertation	4
2	Poisson series and their manipulation	5
2.1	Poisson series	5
2.1.1	Nomenclature and conventions	6
2.1.2	Basic properties	7
2.1.3	Canonical form	7
2.1.4	Fourier series	8
2.2	Poisson series in Celestial Mechanics	8
2.2.1	Example: development of the disturbing function	9
2.2.2	Example: the ELP2000 lunar theory	10
2.3	Term insertion and basic operations on Poisson series	11
2.3.1	Complexity signatures and their effects	11
3	Nontrivial operations on Poisson series	13
3.1	Real powers	13
3.2	Trigonometric operations	16
3.3	Other special functions	18
4	Designing a modern Poisson series manipulator	21
4.1	Why the need for specialized algebraic manipulators?	22
4.1.1	Why the need for another Poisson series manipulator?	22
4.2	Preliminary design considerations	23
4.2.1	Generic programming	24
4.2.2	Choosing a computer language	25
4.2.3	The three lives of Piranha	26
4.3	C++'s features used in Piranha	27
4.3.1	Namespaces	27
4.3.2	Classes	28
4.3.3	Template classes	29
4.3.4	Operator overloading	30
4.3.5	Iterators	31
4.4	The Boost libraries	31

Contents

4.5	Data structures for Poisson series	32
4.5.1	Ordering of terms: binary search trees	32
4.5.2	Identification of terms: hash tables	36
4.5.3	The <code>boost::multi_index_container</code> class	37
5	Piranha: architecture and implementation details	43
5.1	Main classes for Poisson series	43
5.1.1	Example: basic Poisson series coefficient concept	44
5.2	Anatomy of the base series class	45
5.3	Representation of arguments	47
5.4	Toolboxes	47
5.4.1	A note on the implementation of toolboxes	50
5.5	Series I/O	50
5.6	Improving performance	51
5.6.1	Use of temporary hash sets to speed up multiplications	52
5.6.2	Hash function	52
5.6.3	Packed operations on integers and SIMD instructions	53
5.6.4	Memory management	55
5.6.5	Improving evaluation speed	55
5.6.6	Parallelization	56
6	On the manipulation of sparse multivariate polynomials	61
6.1	Types of polynomials	61
6.1.1	Dense polynomials	61
6.1.2	Sparse polynomials	62
6.2	Polynomials in Piranha	63
6.2.1	A general-purpose polynomial class	63
6.3	A faster polynomial class: coded monomial arithmetics	64
6.3.1	Implementation of a sparse polynomial class with coded arithmetics	68
6.4	A mixed approach?	69
7	Pyranha, the Python bindings for Piranha	71
7.1	Easing the utilisation of specific manipulators	72
7.1.1	Issues with existing approaches	72
7.1.2	The Python programming language	73
7.2	Pyranha: brief overview	73
7.2.1	An interactive graphical environment	74
8	Applications	79
8.1	Harmonic development of the TGP	79
8.2	Perturbations in the Saturn planetary system	81
8.2.1	Elliptical orbital elements	82
8.2.2	From elliptical orbital elements to radius	83
8.2.2.1	Eccentricity e	83

8.2.2.2	Complex exponential of M	84
8.2.2.3	Radius r	84
8.2.2.4	Numerical results and limitations	85
9	Future work and performance remarks	87
9.1	Generalising coded arithmetics	87
9.2	Benchmarks	91
9.2.1	Fourier series	91
9.2.2	Multivariate polynomials	93
9.3	Future improvements	95
9.3.1	A more generic architecture?	95
9.3.2	Improving the implementations of data structures	96
9.3.3	SIMD instructions and parallelization	96
9.3.4	Pyranha improvements	96
9.3.5	Interaction with other algebraic manipulators	97
9.4	Availability	97
A	Special functions commonly used in Celestial Mechanics	99
	Bibliography	101
	Nomenclature	109
	Index	111

Contents

LIST OF FIGURES

0.1	Schema di un <code>boost::multi_index_container</code> in cui sono definiti due ordinamenti differenti.	ix
0.2	Gerarchia delle classi nell'architettura di Piranha.	xi
0.3	Uno screenshot di Piranha utilizzato con IPython, matplotlib e PyQt in ambiente GNU/Linux.	xiii
3.1	Testing the precision of real exponentiation of a Fourier series. The blue line represents the inversion of the evaluation of the ELP2000 series ELP3 over a Julian year, the green line represents the absolute value of the difference between the blue line and the evaluation of the inverted ELP3 series.	15
3.2	Testing the precision of the cosine of the Fourier series ELP2 (see explanation in Figure 3.1).	19
3.3	Testing the precision of the associated Legendre function P_2^1 of the lunar series ELP2 (see explanation in Figure 3.1).	20
4.1	Binary search tree (unbalanced).	33
4.2	Balanced binary search tree.	34
4.3	A hash function operation on strings and outputting hexadecimal values.	36
4.4	A hash table storing strings.	37
4.5	Diagram of a <code>boost::multi_index_container</code> with two indices.	38
5.1	Piranha's hierarchy of fundamental classes for the representation of Poisson series.	44
5.2	Base Poisson series class members.	46
5.3	Relation between Piranha's base classes and toolboxes.	49
5.4	Benchmark for three evaluation algorithms: dumb brute force (red), smarter brute force (green) and complex exponential caching algorithm (blue). The series being evaluated are taken from the TASS and ELP2000 theories.	57
5.5	Benchmark of parallelized evaluation. Timings for 300000 evaluations of a Fourier series on two different multi-core system in serial and parallel mode are displayed, and compared to the theoretical minimum running time in parallel mode.	59
7.1	A screenshot of Piranha used in conjunction with IPython and matplotlib in GNU/Linux. The Python prompt is in the bottom-left corner, while on the right the PyQt GUI is displaying the arguments currently defined in the session. Arguments' names can be rendered through a \LaTeX engine, if available. The graph displays the precision of a series multiplication over a timespan, and it has been produced directly from the IPython command line. The graph can be saved in a variety of formats, a capability offered by the matplotlib library.	77

List of Figures

8.1	Titan's orbital radius from TASS (blue line) is compared to its harmonic development obtained through algebraic manipulations. The absolute value of the difference is plotted in green.	86
9.1	Piranha's performance benchmarked against TRIP in a Fourier series multiplication. Piranha's "plain" and "coded" multiplication algorithms are tested with respect to two different versions of TRIP. Running time is expressed in seconds.	92
9.2	Piranha's performance benchmarked against PARI/GP's for a multivariate polynomial multiplication. Piranha is tested using both double precision and arbitrary-size integer coefficients for the polynomials. Running time is expressed in seconds.	94

I INTRODUCTION

ALTHOUGH computer-assisted algebraic manipulation has been employed in many and diverse fields of physics and applied mathematics, its birth and first steps are closely connected to Celestial Mechanics. Perturbative methods, in particular, are naturally fit to be implemented with the aid of an algebraic manipulator, since they involve simple operations (e.g., additions, multiplications, differentiations) on mathematical objects with a simple logical structure (e.g., power series, Fourier series). Since the dawn of the computer era in the '50s, hence, a considerable amount of work in the field of Celestial Mechanics has been devoted to the development of software systems for the automated manipulation of algebraic expressions¹.

This dissertation presents the results of our work on the design and implementation of an algebraic manipulation software specialised for Celestial Mechanics. We refer to a *specialised* algebraic manipulator because its applicability is limited to those mathematical structures commonly employed in perturbation theories, i.e., polynomials and Poisson series. By contrast *general* manipulation systems, like the well-known Maple and Mathematica packages, are built to handle arbitrary mathematical expressions. In specialised manipulators such genericity is sacrificed in favour of performance and lower memory consumption, and as a result their computational efficiency is orders of magnitude better than that of the general systems. Specialised manipulators, in other words, are designed for the solution of large scale problems that are out of reach for general manipulators².

1.1 Motivation

We first came in touch with computer-assisted algebraic manipulation while working on the harmonic development of the tide-generating potential (TGP) in the Earth-Moon-Sun system (see Biscani [2004], Casotto and Biscani [2004a,b]). The initial version of our manipulator was limited to computations on Fourier series, and it was developed because Mathematica turned out to be inadequately slow for the task.

Soon we learnt about the ubiquity of specialised manipulators in Celestial Mechanics and their central role in the formulation of analytical theories of motion. We also learnt about the limitations that afflict the manipulators that have been developed in the last forty years, and which we have identified as follows:

¹When automated calculators were not available, the accurate application of perturbation methods to specific astrodynamical problems could easily take years (see, for instance, Brown's tables of the motion of the Moon - Brown [1919] - or Doodson's harmonical expansion of the tide-generating potential - Doodson [1922]). After the introduction of the first computers, the same tasks could instead be performed in a matter of hours.

²Studies on the long-term stability of the Solar System, for instance, need to manipulate Poisson series of millions of terms (see, for instance, Kuznetsov and Kholshchevnikov [2004]).

1 Introduction

- early manipulators were tied to specific hardware and software architectures and they are unusable on today's computers;
- some of the existing manipulators are *much too specific*: they are written for a limited scope and it is nearly impossible to adapt them to new needs;
- many of the existing manipulators use sub-optimal data structures for the representation of mathematical objects, leading to unsatisfying performance. Performance is crucial and one the *raison d'être* of specialised manipulators;
- source code is generally unavailable, thus it is impossible to re-use existing code or incorporate it in a new software;
- computer languages commonly used for specialised manipulators (FORTRAN and C, mainly) are limiting for the flexibility and reusability of existing code.

The system we have designed and implemented aims at addressing all these points.

1.2 Piranha

The software system we have implemented is called *Piranha*, and it is built to deal principally with objects known as Poisson series. Poisson series can be defined as multivariate Fourier series with multivariate power series as coefficients, and can hence be expressed by the following general formula:

$$\sum_{\mathbf{i}} P_{\mathbf{i}}(x_1, x_2, \dots, x_m) \begin{pmatrix} \cos \\ \sin \end{pmatrix} (i_1 y_1 + i_2 y_2 + \dots + i_n y_n), \quad (1.1)$$

where x and y are literal quantities, $\mathbf{i} = (i_1, i_2, \dots, i_n)$ is a vector of integer values and $P_{\mathbf{i}}$ a multivariate polynomial. Poisson series naturally arise during the application of classical perturbative methods for Celestial Mechanics. The Poisson series form is important because it represents the spectral decomposition of the gravitational disturbing potential, and it allows the application of standard methods for the solution of the equations of motion. A Poisson series manipulator is expected to be able to add, subtract, multiply, differentiate and perform more complicated manipulations on Poisson series while preserving their form.

In designing *Piranha* our main focus has been the search of a fine equilibrium point between performance and genericity. On one hand we wanted to reach top-notch speed of execution (which, as we mentioned earlier, is a major concern in specialised algebraic manipulators); on the other hand we did not want to develop a system which cannot be used for purposes other than those envisioned by the original authors. Most importantly, we wanted *Piranha* to be easily extendible and adaptable by other researchers and developers.

To reach these goals we have drawn from modern computing idioms, data structures and algorithms, thus marking a departure from the programming conventions traditionally associated with algebraic manipulation in Celestial Mechanics. In particular, *Piranha*, which is written in C++, adopts a fully object-oriented (OO) architecture which heavily relies on template programming and meta-programming facilities. This approach is sometimes called *modern C++* (a term

first popularized by Alexandrescu [2001]), and today it constitutes an active area of research in computer science.

The adoption of an OO design, coupled with the flexibility of generic programming through templates, has allowed us to approach the problem of the manipulation of Poisson series from an abstract point of view without any negative impact on performance. The result is that Piranha is not a manipulator: it is a *manipulation framework* that allows its users to define their own manipulators or to extend or modify the already implemented ones with minimal effort³. We see this approach (which was envisioned in a well-known paper by Jacques Henrard - see Henrard [1988]) as a way to address the shortcomings related to the limited scope of the existing manipulators.

Beside the definition of an abstract manipulation framework, with Piranha we have also identified and implemented new methodologies for the manipulation of Poisson series and polynomials. In particular, we have:

- verified that hashed data structures deliver good performance in the context of the manipulation of Poisson series⁴,
- tested and implemented a new and more general methodology for the calculation of circular functions of Poisson series (based on the Jacobi-Anger developments),
- identified and implemented a methodology for the multiplication of sparse multivariate polynomials and Poisson series which is based on the Kronecker algorithm and whose performance in practice is promising. As far as we were able to verify from the literature, it is the first time that the Kronecker algorithm is applied to the manipulation of Poisson series.

We have also verified that Piranha can compete, performance-wise, on the same level of more mature and optimized software, hence showing that the level of abstractness allowed by the framework is not detrimental for performance.

In designing Piranha we have strived to employ as much as possible existing libraries and solutions for common computational problems. The implementations of the basic data structures used in Piranha, for instance, are those available in the standard C++ library. As result, Piranha is built on a stack of Free Software libraries, and in particular it relies heavily on the facilities provided by the Boost C++ libraries (see Boost). This choice has proven to be effective, since from the beginning we could rely on a very solid starting point without the need to “reinvent the wheel”. Although the standard implementations of algorithms and data structures have proven to be adequate, there is always the possibility to re-implement them in a more optimized way if necessary. Thanks to the generic nature of C++ template programming, implementations can be swapped out without the need to change any other part of the code.

Piranha has been presented in various international meetings dedicated to Celestial Mechanics, both as the main subject (Biscani and Casotto [2006], Casotto and Biscani [2007, 2008]) and in the context of other works (Casotto and Biscani [2005]). We expect that, as its capabilities

³ At the time of this writing Piranha implements 12 different manipulators in ~12000 lines of code.

⁴Hashed containers are strangely uncommon in the landscape of specific manipulators for Celestial Mechanics, although their use is more widespread in other fields.

1 Introduction

grow, Piranha will acquire an increasingly relevant role in our research projects, especially in the context of methods for perturbation theories.

Work on Piranha will continue towards the implementation of the functionalities still missing and towards the optimization of the existing algorithms. Piranha will soon be made available under a Free Software license and we will try to build a community of users and developers around it.

1.3 Structure of the dissertation

In the first part of this dissertation (Chapters 2 and 3) we introduce Poisson series by showing where they arise from in Celestial Mechanics and providing a formal definition. We introduce the concept of canonical form for Poisson series and show the basic mathematical operations that can be performed on them. We also show how we can perform common nontrivial operations on Poisson series (e.g., circular functions, real powers) by means of truncated expansions.

In Chapter 4 we introduce the preliminary design considerations for Piranha that stem from the analysis of the Poisson series structure described earlier and from the study of the literature. We explain the choice of the C++ programming language and introduce template programming. We then proceed to describe C++'s features used in Piranha and the Boost libraries. We also introduce the concepts of binary search trees and hash tables, and show how they can be efficiently used in the manipulations of Poisson series.

Chapter 5 examines in deeper detail the manipulation framework defined by Piranha. C++ concepts and models are introduced, and the main classes of the framework are briefly described. The concept of toolbox is also explained, and some techniques to speed up Piranha during the most demanding operations are presented.

Chapter 6 deals with the manipulation of sparse multivariate polynomials. It is shown how these entities can be treated similarly to Fourier series. An application of the Kronecker algorithm to speed up the multiplication of sparse multivariate polynomials is presented and explained.

In Chapter 7 Pyranha, the Python bindings for Piranha, are presented. With Pyranha it is possible to access Piranha's capabilities from the popular Python language without sacrificing performance. It is also shown how Pyranha can be used to provide an interactive graphical access to Piranha.

In Chapter 8 two real-world applications of Piranha are presented. The first one shows how the tide-generating potential in the Sun-Earth-Moon system can be harmonically decomposed with the help of Piranha; the second one shows how to transform the theory of motion for Saturn's satellites, TASS, from one coordinate system to another.

Finally, in Chapter 9, Kronecker's algorithm is applied also to Poisson series, and Piranha is benchmarked against two popular manipulation packages. Our tests show how Piranha can compete with more mature and optimized software. Future work is also presented.

2 POISSON SERIES AND THEIR MANIPULATION

POISSON series (Danby et al. [1966]) have a central role in both classical and modern Celestial Mechanics. In a sense it could be said that Poisson series constitute one of the basic mathematical building blocks of perturbation theories: they unify polynomials, which typically arise from Taylor expansions, and Fourier series, which are the logical choice when dealing with time-periodic phenomena. In this chapter we will define Poisson series, show where they arise from in Celestial Mechanics and illustrate the basic operations that can be performed on them.

2.1 Poisson series

For the purpose of this dissertation we will adopt the definition of Poisson series given in San-Juan and Abad [2001]:

Definition Poisson series are multivariate Fourier series with multivariate Laurent series as coefficients.

We recall here the definition of Laurent series:

Definition Laurent series are power series with complex coefficients which include terms of negative degree.

Although in Celestial Mechanics Poisson series originate from infinite series, in practice these series are truncated to include a finite number of terms. From now on when referring to Poisson series, hence, we will mean “finitely truncated multivariate Fourier series with finitely truncated multivariate Laurent series as coefficients”.

The formula of a Poisson series is then:

$$\sum_{\mathbf{i}} \sum_{\mathbf{j}} M_{\mathbf{j}}(x_1, x_2, \dots, x_m) \begin{pmatrix} \cos \\ \sin \end{pmatrix} (\mathbf{i}_1 y_1 + \mathbf{i}_2 y_2 + \dots + \mathbf{i}_n y_n). \quad (2.1)$$

\mathbf{i} and \mathbf{j} are vectors of integers whose sizes are n and m respectively, so that

$$\mathbf{i} = (i_1, i_2, \dots, i_n) : i_k \in \mathbb{Z} \forall k \in [1, n], \quad (2.2)$$

$$\mathbf{j} = (j_1, j_2, \dots, j_m) : j_k \in \mathbb{Z} \forall k \in [1, m]. \quad (2.3)$$

2 Poisson series and their manipulation

M_j is a multivariate monomial in the literal quantities (x_1, x_2, \dots, x_m) with integer exponents and complex numerical coefficient C_j :

$$M_j = C_j x_1^{j_1} x_2^{j_2} \dots x_m^{j_m}. \quad (2.4)$$

It is important to note that in eq. (2.1) it is implied that every i vector has its own set of j vectors, so that the quantity

$$P_i(x_1, x_2, \dots, x_m) = \sum_j M_j(x_1, x_2, \dots, x_m) \quad (2.5)$$

is a complex multivariate polynomial with integer exponents, and eq. (2.1) can be rewritten more compactly as

$$\sum_i P_i(x_1, x_2, \dots, x_m) \begin{pmatrix} \cos \\ \sin \end{pmatrix} (i_1 y_1 + i_2 y_2 + \dots + i_n y_n). \quad (2.6)$$

The literal quantities (x_1, x_2, \dots, x_m) are known as the *polynomial arguments* of the Poisson series, while the literal quantities (y_1, y_2, \dots, y_n) are known as the *trigonometric arguments*.

2.1.1 Nomenclature and conventions

From now on we will adopt the following conventions (with reference to eqs. (2.1) and (2.6)):

- n will be referred to as the *trigonometric width* of the series,
- m will be referred to as the *polynomial width* of the series,
- the number of terms of a Poisson series will be called the *length* of the series,
- the quantities $P_i(x_1, x_2, \dots, x_m)$ and

$$\begin{pmatrix} \cos \\ \sin \end{pmatrix} (i_1 y_1 + i_2 y_2 + \dots + i_n y_n)$$

will be referred to respectively as the *polynomial part* (or, more generally, *coefficient*) and *trigonometric part* of the terms of a Poisson series,

- except when explicitly stated otherwise, in order to make the notation less cumbersome the appearance of “cos” inside the formula of a Poisson series will mean that the trigonometric function can be either cosine or sine.

Additionally, we will keep referring to the indices vectors of trigonometric and polynomial arguments as the i -vectors and j -vectors respectively. The elements of the i -vectors will also be referred to as *trigonometric multipliers*, while the elements of the j -vectors will also be referred to as *polynomial exponents*.

2.1.2 Basic properties

It is trivial to show that the set of Poisson series constitutes an abelian group under the operations of addition (+) and subtraction (−), and whose identity element is the *null* Poisson series with the following characteristics:

- the number of polynomial and trigonometric arguments is zero,
- there is just one term in the series and this term has a numerical coefficient C_j equal to zero.

Additionally, the set of Poisson series is closed under the operation of multiplication (·) and partial derivation with respect to polynomial and trigonometric arguments. When multiplying two Poisson series the products of trigonometric parts are reduced to combinations of cosines and sines through Werner's trigonometric formulas:

$$\begin{aligned}
 A \cos \alpha \cdot B \cos \beta &= \frac{AB}{2} \cos(\alpha - \beta) + \frac{AB}{2} \cos(\alpha + \beta), \\
 A \cos \alpha \cdot B \sin \beta &= \frac{AB}{2} \sin(\alpha + \beta) - \frac{AB}{2} \sin(\alpha - \beta), \\
 A \sin \alpha \cdot B \cos \beta &= \frac{AB}{2} \sin(\alpha - \beta) + \frac{AB}{2} \sin(\alpha + \beta), \\
 A \sin \alpha \cdot B \sin \beta &= \frac{AB}{2} \cos(\alpha - \beta) - \frac{AB}{2} \cos(\alpha + \beta).
 \end{aligned} \tag{2.7}$$

Poisson series, however, do not constitute an abelian group under the operation of multiplication, since, in general, the inverse of a Poisson series cannot be expressed as another Poisson series in finite terms (a procedure to compute the approximated inverse of a Poisson series is illustrated in §3.1).

2.1.3 Canonical form

The notion of a canonical form for Poisson series, while not strictly necessary from a mathematical point of view, is extremely helpful for the representation of Poisson series in computer algebra systems.

Definition A Poisson series is said to be in *canonical form* when the following conditions are met:

1. the first non-zero element of all the i -vectors is strictly positive¹,
2. the set of i -vectors contains no duplicate elements,
3. there are no terms with $C_j = 0$,
4. there are no terms with sine trigonometric part and null i -vector.

¹This is just a convention, since an equally useful canonical form could be defined with all the first non-zero elements being strictly negative.

2 Poisson series and their manipulation

This canonical form ensures that the number of terms needed to represent a Poisson series is kept to the minimum, and allows to simplify the algorithms employed in the manipulations. From now on it will be assumed that all the Poisson series, unless otherwise specified, are in canonical form.

2.1.4 Fourier series

A useful subset of Poisson series which typically appears in the solutions of the theories of motion of celestial bodies is the one characterized by a null polynomial width:

$$\sum_i C_i \cos(i_1 y_1 + i_2 y_2 + \dots + i_n y_n). \quad (2.8)$$

The polynomial parts of the terms, in other words, have all degree 0 and are hence purely numerical. This kind of Poisson series is often referred to in the literature as *Fourier series*. In this dissertation we will adopt the same convention.

Additionally, since Piranha was initially conceived for the manipulation of Fourier series and because Poisson series can be treated with the same techniques adopted for Fourier series, we will often use Fourier series in examples and benchmarks.

2.2 Poisson series in Celestial Mechanics

Poisson series in Celestial Mechanics arise in the early stages of the formulation of a theory of motion², and can be seen as one of the fundamental mathematical building blocks of perturbative methods. The Poisson series form is sought after and maintained throughout the machinery of perturbation theories because it provides the spectral decomposition of the disturbing gravitational potential. This kind of decomposition is fundamental because it allows to discriminate the type of perturbation and hence to apply the standard methods of solution of the equations of motion.

A researcher interested in the long-period evolution of the Solar System, for instance, is typically interested in the secular (i.e., slowly varying) perturbative terms of the disturbing function and discards the high-frequency terms assuming that their contribution over a sufficiently long time span averages to zero. In Murray and Dermott [2000], Chapter 6, for instance, it is shown how this approach, an application of the *averaging principle*, allows to devise an analytical description of the long-time evolution of Jupiter's orbital elements in good accordance with numerical integrations. More generally, the spectral decomposition of the disturbing potential allows to classify the perturbative contributions and keep only those which are relevant to the particular problem being considered.

Not only the Poisson series form is kept throughout the calculations of perturbative theories: theories of motion themselves are expressed as Poisson series. High precision analytical theories like VSOP87 (Bretagnon and Francou [1988]) for the planets of the Solar System, ELP2000 (Chapront-Touzé and Chapront [1988]) for the Earth-Moon system, TASS (Vienne and Duriez

²We recall that the term *theory of motion* in this context refers to an analytical formula expressing the evolution in time of the position of a celestial body.

[1995]) for Saturn's main satellites and theories of lunar libration (e.g., Moons [1982]) are all expressed as Poisson series. The Poisson series form of these theories allows them to be used as starting points for other studies, as seen for instance in Casotto and Biscani [2004a] to achieve the harmonic decomposition of the tide-generating potential.

2.2.1 Example: development of the disturbing function

One of the first steps in the formulation of a theory of motion is the development of the disturbing function, i.e., the gravitational attraction exercised by the secondary bodies in a system of self-gravitating masses dominated by the gravitational pull of one massive primary body. Following the procedure and the notation of Murray and Dermott [2000], Chapter 6, the disturbing function R on the internal secondary in a three-body coplanar system is first expanded into Legendre polynomials P_l :

$$R \propto \frac{1}{r'} \sum_{l=2}^{\infty} \left(\frac{r}{r'} \right)^l P_l(\cos \psi), \quad (2.9)$$

where r and r' , with $r < r'$, are the two secondaries' distances from the primary and ψ is their angular separation. Legendre polynomials can be expressed as natural powers of their arguments $\cos \psi$, which in turn can be expressed in terms of the secondaries' true anomalies and longitudes of periapsis, f , f' , ϖ and ϖ' respectively:

$$\cos \psi = \cos (f' - f + \varpi' - \varpi). \quad (2.10)$$

The cosines and sines of the true anomalies can then be extracted from $\cos \psi$ through elementary trigonometric formulas, and expanded using the well-known elliptic expansions (see Murray and Dermott [2000], Chapter 2)

$$\cos f = -e + \frac{2(1-e^2)}{e} \sum_{s=1}^{\infty} J_s(se) \cos sM, \quad (2.11)$$

$$\sin f = 2\sqrt{1-e^2} \sum_{s=1}^{\infty} \frac{1}{s} \frac{d}{de} J_s(se) \sin sM, \quad (2.12)$$

where e and M have the usual meanings of eccentricity and mean anomaly, and J_s are Bessel functions of the first kind (see Abramowitz and Stegun [1964]). If the Bessel functions are expressed as power series, their derivatives calculated and the quantities $1/e$ and $\sqrt{1-e^2}$ expanded through a MacLaurin development, the Poisson series form is achieved. For instance, the expansion for $\cos f$ up to the fourth order in eccentricity is

$$\begin{aligned} \cos f = & \cos M + e(\cos 2M - 1) + \frac{9}{8}e^2(\cos 3M - \cos M) \\ & + \frac{4}{3}e^3(\cos 4M - \cos 2M) \\ & + e^4 \left(\frac{25}{192} \cos M - \frac{225}{128} \cos 3M + \frac{625}{384} \cos 5M \right). \end{aligned} \quad (2.13)$$

2 Poisson series and their manipulation

This is a Poisson series in non-canonical form with trigonometric and polynomial widths equal to one. It is then possible to expand in a similar fashion the quantities $1/r'$ and r/r' in eq. (2.9) and obtain again Poisson series in polynomial variables e and e' and trigonometric variables M and M' . Since the set formed by Poisson series is closed under the operations of addition and multiplication, the expression for the disturbing function R is again a Poisson series.

We would like to point out the fast growth of the number of Poisson series terms resulting even from this simple calculation. By taking into account a non-coplanar physical model, by pushing the developments to higher orders and through the successive manipulations that are usually performed on the disturbing function, the lengths of the series being manipulated grow explosively. Modern studies on the long-term evolution of planetary systems employ series of millions of terms (see, for instance, Kuznetsov and Kholoshevnikov [2004]).

2.2.2 Example: the ELP2000 lunar theory

The ELP2000 lunar theory (see Chapront-Touzé and Chapront [1988]) provides an accurate analytical description of the motion of the Earth-Moon system around the Sun which, in its most complete form, is characterised by an accuracy comparable to that of numerical integrations (see, for instance, Chapront and Chapront-Touzé [1981]). As an illustrative example, we reproduce here the first terms of the solution of the main problem³ of the lunar theory for the distance of the Moon from Earth's barycenter:

Amplitude (km)	D	l'	l	F
385000.52719	0	0	0	0
-20905.32206	0	0	1	0
-3699.10468	2	0	-1	0
-2955.96651	2	0	0	0
...

This table translates into the Fourier series

$$385000.52719 - 20905.32206 \cos l + \\ - 3699.10468 \cos (2D - l) - 2955.96651 \cos 2D + \dots \quad (2.14)$$

The first term, a constant, is roughly equivalent to the mean lunar distance and represents an unpertrubed circular motion. The following terms express the deviation from the circular orbit. The literal quantities D , l' , l and F are called *Delaunay arguments*, and they are expressed as power series in time according to the general formulation:

$$\lambda = \lambda^{(0)} + \lambda^{(1)}t + \lambda^{(2)}t^2 + \lambda^{(3)}t^3 + \lambda^{(4)}t^4, \quad (2.15)$$

where the $\lambda^{(i)}$ have known numerical values. To compute the lunar distance at a certain time t_0 it is sufficient to evaluate the Delaunay arguments at $t = t_0$, substitute such numerical evaluation

³The term *main problem of the lunar theory* designates a simplified dynamical model in which Earth, Sun and Moon are considered point masses subject only to their mutual gravitational attraction (see Brown [1960]).

into eq. (2.14) and evaluate the trigonometric functions and the summation of the terms of the series.

The full solution for the Earth-Moon-Sun system provided by the ELP2000 theory includes figure perturbations, planetary perturbations, tidal effects and relativistic perturbations, and is comprised of approximately 37000 terms.

2.3 Term insertion and basic operations on Poisson series

As stated in §2.1.2, the Poisson series set is closed under the operations of addition/subtraction and multiplication. Both addition/subtraction and multiplication are performed through the fundamental operation on Poisson series, i.e., the insertion of a term. Series addition, indeed, is merely an insertion of all the terms of one series into the other one (a sign change is required when performing a subtraction), while series multiplication is the insertion into an empty series of terms generated using Werner's trigonometric formulas.

Term insertion, in turn, is built upon the capability of identifying a term in the series. In order to preserve the canonical form of the series, indeed, it is necessary to be able to discern if a term with the same trigonometric part of the term being inserted already exists in the series. If this is the case, then the term won't be inserted in the series: its coefficient will instead be added (or subtracted) to the one belonging to the term already existing in the series. We refer to this operation as *term packing*.

The algorithm for the insertion of a term T into a Poisson series is then the following:

1. if T's coefficient is zero, discard the term;
2. if T's i-vector is null and the trigonometric part is a sine, discard the term;
3. if T's i-vector's first non-null element is less than zero, invert the sign of all the elements of the vector. Invert also the sign of the coefficient if the trigonometric part is a sine;
4. identify T's trigonometric part among the terms of the series: if there exists a match, update the matching term's coefficient by adding (or subtracting) T's, otherwise append T to the series;
5. if a coefficient update took place, verify that the modified coefficient is not zero. If it is, delete the corresponding term from the series.

This algorithm will preserve the canonical form of Poisson series.

2.3.1 Complexity signatures and their effects

Given two Poisson series P_1 and P_2 , with lengths respectively l_1 and l_2 , it is evident that both the addition and subtraction of P_1 and P_2 require

$$\min [l_1, l_2] \tag{2.16}$$

2 Poisson series and their manipulation

term insertions. The resulting series will have a maximum length of

$$l_1 + l_2 \tag{2.17}$$

terms (the actual length will depend on the packing ratio of P_1 's terms with respect to P_2 's). Using the big O notation (see Knuth [1998a]) we can say that Poisson series addition and subtraction have $O(\min [l_1, l_2])$ runtime complexity and $O(l_1 + l_2)$ memory storage requirement. In other words, the complexity signatures are linear both in processing power and memory utilization.

By contrast, the multiplication operation is much more demanding, and requires

$$2l_1 l_2 \tag{2.18}$$

term insertions (the factor of two appearing because for each term-by-term multiplication two terms are generated as per eqs. (2.7)) and the same number of coefficient multiplications. The size of the resulting series will also be in the order of $2l_1 l_2$. The complexity signatures for series multiplication are hence quadratic, or $O(l_1 l_2)$.

These complexity signatures lead to two very important considerations:

1. series multiplication is by far the most demanding operation to be performed in the manipulation of Poisson series, both with respect to processing power and memory usage. An efficient Poisson series manipulator is fast at performing series multiplications;
2. it is necessary to be able to truncate series during multiplications.

To have an idea of the problems related to the quadratic growth in the number of terms when multiplying Poisson series it is sufficient to see what happens with three series with 1000 terms each: after their multiplication a series with a maximum number of 4 billions terms will be generated. It is then necessary to implement *truncation methodologies* in the multiplication of Poisson series. Such methodologies will depend on the problem being considered and on the type of series being manipulated. For instance:

- when manipulating Fourier series it is convenient to adopt a truncation criterion based on the absolute value of the coefficients (which, by definition, are purely numerical);
- when dealing with manipulations in perturbation theories the usual truncation criterion is based on the degree of the polynomial coefficients or on the degree of one particular literal symbol.

Whatever methodology is chosen, the need to truncate typically translates into the need to establish an ordering on the terms of a Poisson series which allows to skip all term-by-term multiplications from a certain point onward.

These considerations are crucial in the design of an efficient Poisson series manipulator, and impose certain constraints and requirements on the data structures to be employed in such a software.

3 NONTRIVIAL OPERATIONS ON POISSON SERIES

IN Celestial Mechanics we are typically interested in performing mathematical operations on Poisson series which are less trivial than those described in the previous chapter. Some examples include:

- trigonometric functions (complex exponential, sin and cos),
- real powers, particularly inversion and square root,
- differential operators.

On top of such nontrivial operations other capabilities can be built, like special functions relevant to Celestial Mechanics (e.g., Legendre polynomials and Legendre functions, spherical harmonics, rotation and translation theorems for spherical harmonics – see Appendix A) having Poisson series as arguments.

For the purposes described in §2.2 we are interested in maintaining the Poisson series form throughout the calculations. However we have seen that the set formed by Poisson series is closed only under the operations of addition/subtraction, multiplication and partial derivation with respect to the arguments. It follows then that we will need to express the desired nontrivial manipulations on Poisson series in terms of additions and multiplications, typically through truncated series expansions.

In this chapter we are going to show some of the series expansions we can use to perform nontrivial operations on Poisson series. The capabilities described here are thoroughly used in the applications shown in Chapter 8.

3.1 Real powers

Real powers, and especially inversion and square root, are used thoroughly in the context of Celestial Mechanics. Typically they arise from the dependence of the gravitational force from the inverse of the square of the distance.

In Piranha we have implemented exponentiation to real powers for Poisson series through the application of the binomial theorem (Abramowitz and Stegun [1964], Arfken and Weber

3 Nontrivial operations on Poisson series

[2005]):

$$\begin{aligned} \left[\sum_{\mathbf{ij}} C_{\mathbf{ij}} \cdot x_1^{j_1} x_2^{j_2} \cdot \dots \cdot x_m^{j_m} \cdot \begin{pmatrix} \cos \\ \sin \end{pmatrix} (\mathbf{i} \cdot \mathbf{a}) \right]^\alpha &= \\ &= (A + X)^\alpha = \sum_{k=0}^{\infty} \binom{\alpha}{k} X^k A^{\alpha-k} = \\ &= A^\alpha \sum_{k=0}^{\infty} \binom{\alpha}{k} \left(\frac{X}{A} \right)^k, \end{aligned} \quad (3.1)$$

where in our case A is the leading term of the series and X represents the rest of the series. In order to be raised to a real power a Poisson series must respect such constraints:

1. the leading term must lend itself to real exponentiation (as we need to calculate $1/A$ and A^α , as shown in eq. (3.1)),
2. the leading term's coefficient must outweigh all other coefficients combined together.

The first condition is always met when dealing with Fourier series, while in the case of polynomial coefficients the binomial theorem can be applied recursively to calculate $1/A$ and A^α . The second condition is needed both to make sure that there are no singularities in the resulting series when calculating negative powers and to respect the convergence criterion of the binomial theorem. If this condition is met, the series' evolution in time is an oscillation around the value of the leading coefficient, and the series never evaluates to zero. A variant of this technique has been described in Broucke [1971]. Since $\alpha \in \mathfrak{R}$, the generalized formulation for the binomial coefficient must be used in eq. (3.1):

$$\binom{\alpha}{k} = \frac{\alpha(\alpha-1)(\alpha-2) \cdot \dots \cdot (\alpha-k+1)}{k!} = \frac{(\alpha)_k}{k!}, \quad (3.2)$$

where $(\cdot)_k$ is the notation for the Pochhammer symbol, also known as "falling factorial" (see Knuth [1992]). The error Δ introduced by stopping the development at order n_0 is then

$$\Delta = A^\alpha \sum_{k=n_0+1}^{\infty} \binom{\alpha}{k} \left(\frac{X}{A} \right)^k. \quad (3.3)$$

Δ is then strictly dependent upon X/A : the smaller X is with respect to A , the faster the development converges.

The constraints forced by the application of the binomial theorem are typically not limiting in Celestial Mechanics, since inversion and square roots will most likely be applied to the radii of actual perturbed orbits in which singularities are not considered.

In Figure 3.1 we have plotted a precision test for the inversion of the Fourier series ELP3 representing the lunar distance in the ELP2000 lunar theory performed with our manipulator, Piranha (see Chapters 4 and 5). The blue line represents a historical series produced by inverting

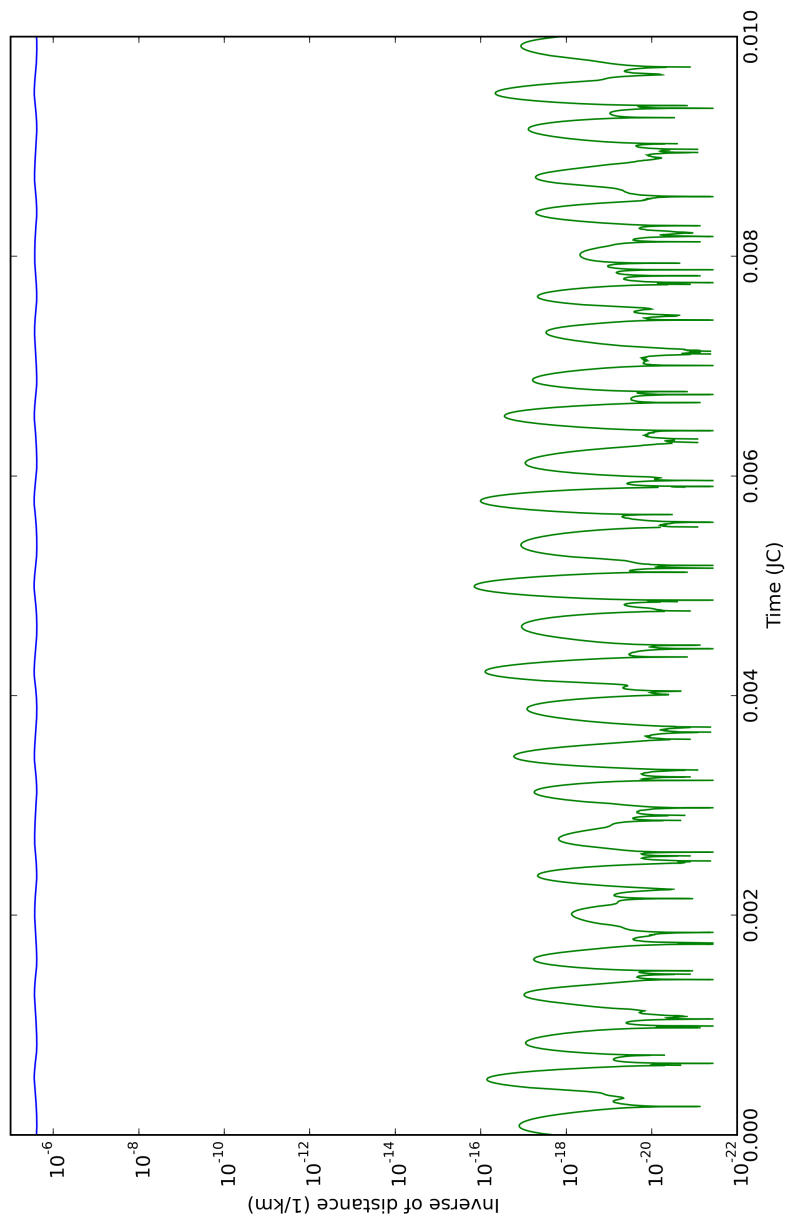


Figure 3.1: Testing the precision of real exponentiation of a Fourier series. The blue line represents the inversion of the evaluation of the ELP2000 series ELP3 over a Julian year, the green line represents the absolute value of the difference between the blue line and the evaluation of the inverted ELP3 series.

3 Nontrivial operations on Poisson series

the evaluation of ELP3 over a timespan of one Julian year, while the green line represents the difference, in absolute value, between the blue line evaluation and the evaluation of the series calculated with the application of the binomial theorem. The truncation threshold was set to one part every ten millions. It can be noted how the application of the binomial expansion method leads to an accurate representation for the inverse of ELP3 (the precision can be furtherly improved by lowering the truncation threshold)¹.

3.2 Trigonometric operations

Another class of operations often performed on Poisson series (and especially on Fourier series) is the one comprising trigonometric special functions. In Piranha cosine and sine of Poisson series are expanded by applying the Jacobi-Anger developments on each term of the series (see Brown and Churchill [1993] and Weisstein [2007]):

$$e^{ix \cos \theta} = \sum_{n=0}^{\infty} (2 - \delta_{0n}) (-1)^n J_{2n}(x) \cos(2n\theta) + 2i \sum_{n=0}^{\infty} (-1)^n J_{2n+1}(x) \cos[(2n+1)\theta], \quad (3.4)$$

$$e^{ix \sin \theta} = \sum_{n=0}^{\infty} (2 - \delta_{0n}) J_{2n}(x) \cos(2n\theta) + 2i \sum_{n=0}^{\infty} J_{2n+1}(x) \sin[(2n+1)\theta], \quad (3.5)$$

where the J_n are Bessel functions of the first kind, which can be expressed by the following power series:

$$J_n(x) = \sum_{l=0}^{\infty} \frac{(-1)^l}{2^{2l+n} l! (n+l)!} x^{2l+n}. \quad (3.6)$$

By using these developments it is possible to express the complex exponential of a series as

$$e^{i(\sum_{ij} C_{ij} \cdot x_1^{j_1} x_2^{j_2} \dots x_m^{j_m} \cdot \cos(\mathbf{i} \cdot \mathbf{a}))} = \prod_{ij} \left\{ \sum_{n=0}^{\infty} (2 - \delta_{0n}) (-1)^n J_{2n} \left(C_{ij} \cdot x_1^{j_1} x_2^{j_2} \dots x_m^{j_m} \right) \cos[2n(\mathbf{i} \cdot \mathbf{a})] + 2i \sum_{n=0}^{\infty} (-1)^n J_{2n+1} \left(C_{ij} \cdot x_1^{j_1} x_2^{j_2} \dots x_m^{j_m} \right) \cos[(2n+1)(\mathbf{i} \cdot \mathbf{a})] \right\}, \quad (3.7)$$

where we have assumed that all terms are cosines for simplicity. By taking the real (resp. imaginary) part of this product, the cosine (resp. sine) of a Poisson series can then be computed. We

¹We also note that this and the following graphs were produced interactively entirely from Piranha, the Python bindings for Piranha (see Chapter 7).

note that, since we can express Bessel functions of the first kind as power series and remembering that the set formed by Poisson series is closed under the operation of multiplication, the right hand side of eq. (3.7) represents a Poisson series whose coefficients are polynomials with rational coefficients.

By using the definition of Bessel functions and remembering the convergence properties of infinite geometric series, an upper limit for the error resulting from the truncation of the infinite Jacobi-Anger developments under certain conditions can be found. We start by noting that the sums of the series in eqs. (3.4) and (3.5), by exhibiting alternating signs and because of the presence of trigonometric parts in the terms, are always less than or equal to those of the series whose terms are the absolute values of Bessel functions of even or odd order:

$$\sum_{n=0}^{\infty} |J_{2n}(x)|, \quad (3.8)$$

$$\sum_{n=0}^{\infty} |J_{2n+1}(x)|. \quad (3.9)$$

By remembering the definition of Bessel function of the first kind as power series, the following (in)equalities are then easily found:

$$\sum_{n=0}^{\infty} |J_{2n}(x)| = \sum_{n=0}^{\infty} \left| \sum_{l=0}^{\infty} \frac{(-1)^l}{2^{2l+2n} l! (2n+l)!} x^{2l+2n} \right| \quad (3.10)$$

$$\leq \sum_{n=0}^{\infty} \sum_{l=0}^{\infty} \left(\frac{|x|}{2} \right)^{2l+2n} \frac{1}{l! (2n+l)!} \quad (3.11)$$

$$= \sum_{l=0}^{\infty} \left(\frac{|x|}{2} \right)^{2l} \left[\sum_{n=0}^{\infty} \left(\frac{|x|}{2} \right)^{2n} \frac{1}{l! (2n+l)!} \right] \quad (3.12)$$

$$\leq \sum_{l=0}^{\infty} \left(\frac{|x|}{2} \right)^{2l} \left[\sum_{n=0}^{\infty} \left(\frac{|x|}{2} \right)^{2n} \right] \quad (3.13)$$

$$= \sum_{l=0}^{\infty} \left(\frac{x^2}{4} \right)^l \sum_{n=0}^{\infty} \left(\frac{x^2}{4} \right)^n. \quad (3.14)$$

Similar relations hold for odd orders. For $|x| < 2$ the two geometric series in eq. (3.14) converge to the well-known value,

$$\frac{4}{4-x^2}, \quad (3.15)$$

hence leading to the inequality

$$\sum_{n=0}^{\infty} |J_{2n}(x)| \leq \left(\frac{4}{4-x^2} \right)^2. \quad (3.16)$$

By replaying this procedure after substituting n with $n + n_f + 1$ in the terms of the series, we can find an upper bound for the error induced by the truncation of the series in the real parts of

3 Nontrivial operations on Poisson series

eqs. (3.4) and (3.5) at $n = n_f$, which we call $e_{n_f}^r$:

$$|e_{n_f}^r| \leq 2 \left(\frac{x^2}{4}\right)^{n_f+1} \left(\frac{4}{4-x^2}\right)^2. \quad (3.17)$$

A similar procedure leads to the following truncation error for the imaginary parts:

$$|e_{n_f}^i| \leq |x| \left(\frac{x^2}{4}\right)^{n_f+1} \left(\frac{4}{4-x^2}\right)^2. \quad (3.18)$$

We note here that in the context of Celestial Mechanics the condition $|x| < 2$ is not limiting, since in most cases the coefficients of Poisson series subject to complex exponentiation will be natural powers of fractions of the eccentricity and/or sines of the inclination, and as such they will respect this condition. As an example, we note here that the largest amplitude in the series expressing the perturbation on the Moon's longitude in the ELP2000 theory amounts to ~ 0.11 rad.

Sine and cosine of Poisson series have already been implemented in some of the existing manipulators for Celestial Mechanics. However, to our knowledge, such operations have until now been implemented using Taylor series, and hence assuming that the series are somehow separable into a dominant contribution and a small parameter (e.g., Broucke [1970] and Chapront [2003b,a]). By using the Jacobi-Anger developments it is instead possible to calculate the complex exponential of a wider class of Poisson series.

Figure 3.2, analogously to Figure 3.1, shows a precision test on the cosine of the ELP2 series representing the lunar colatitude for the main problem of the lunar theory ELP2000. The truncation level is set to one part every ten millions.

3.3 Other special functions

With the availability of complex exponentiation and real powers it is possible to have Poisson series as arguments of other special functions. The most relevant in the context of Celestial Mechanics are probably associated Legendre functions and spherical harmonics (see Appendix A for definitions), which are used pervasively in the application of perturbative methods.

For the calculation of associated Legendre functions well known recurrence relations can be used (see Abramowitz and Stegun [1964], for instance), while spherical harmonics and their solid counterparts are calculated from Legendre functions multiplied by real powers and complex exponentials of Poisson series.

Other useful applications in Celestial Mechanics involve the transformation of such special functions, like addition, rotation and translation theorems for Legendre functions and spherical harmonics.

Figure 3.3 shows a precision test for the calculation of the associated Legendre function of degree two and order one of the ELP2000 series ELP2. The truncation level is set to one part every 10^{10} .

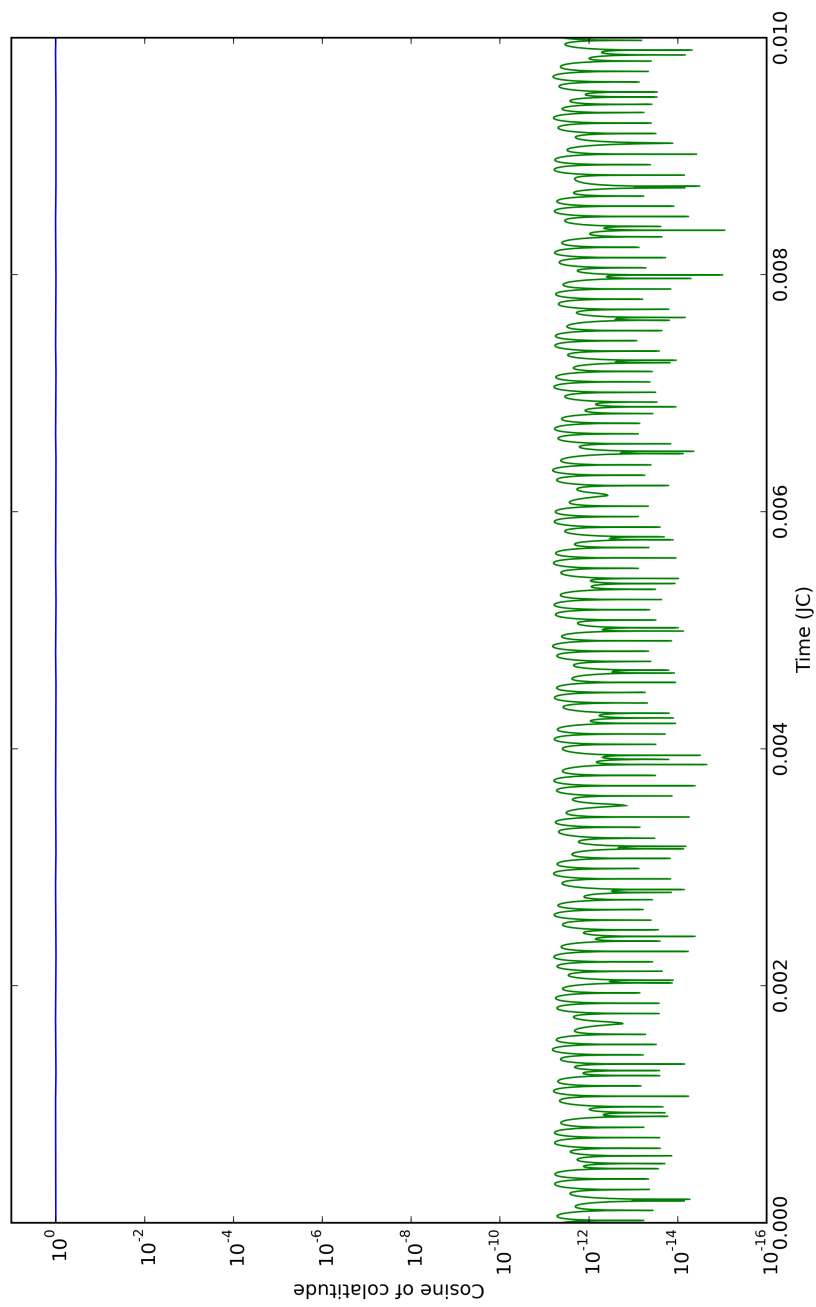


Figure 3.2: Testing the precision of the cosine of the Fourier series ELP2 (see explanation in Figure 3.1).

3 Nontrivial operations on Poisson series

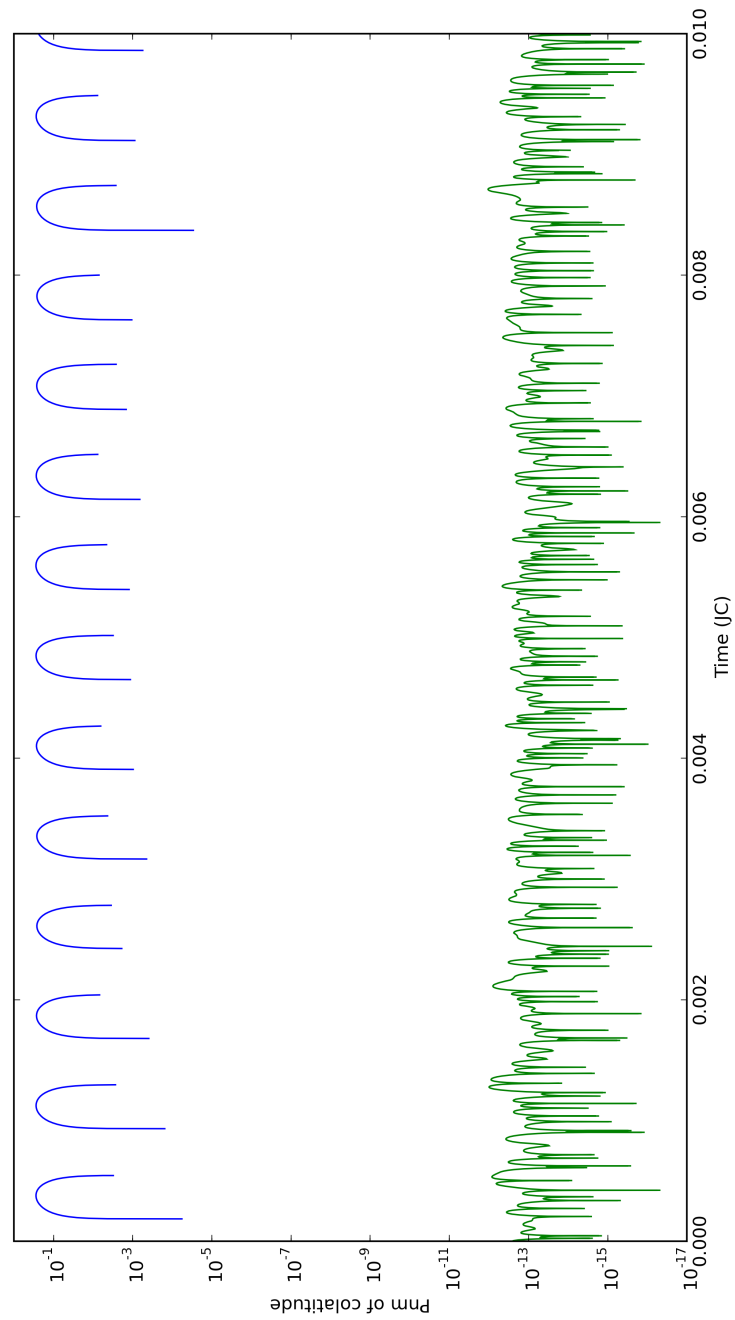


Figure 3.3: Testing the precision of the associated Legendre function P_2^1 of the lunar series ELP2 (see explanation in Figure 3.1).

4 DESIGNING A MODERN POISSON SERIES MANIPULATOR

If you have four hours to cut down a tree, then you should pass the first three hours sharpening your axe.

– *Old Swedish saying*

POISSON series manipulators have been developed since the dawn of the computer era, and as of today a great number of manipulators exists. Since Herget and Musen [1959] many authors have developed their own algebraic manipulators, and it does not come as a surprise that many of the first mainframes were bought by astronomy departments with algebraic manipulation in mind.

Among the most important Poisson series manipulators, without claims of completeness, we recall Broucke and Garthwaite [1969], Jefferys [1970, 1972], Rom [1970], Bourne and Horton [1971], Babaev et al. [1980], Dasenbrock [1982], Richardson [1989], Abad and San-Juan [1994], Ivanova [1996], Chapront [2003b,a] and Gastineau and Laskar [2005]. Henrard [1988] and Laskar [1990] have analyzed the data structures and the algorithms commonly employed in Poisson series manipulators, and, despite their age, these papers still provide a good overview of the issues to overcome to write a good manipulator. Specific manipulators for Celestial Mechanics have been written also for mathematical entities different from Poisson series, like echeloned Poisson series (see Ivanova [2001]) and Kinoshita series (see Navarro and Ferrándiz [March 2002]).

The sophistication of Poisson series manipulators has increased together with the exponential growth of computing power in the last decades. While the first manipulators supported only elementary operations on fixed-width series, today's manipulators are extremely versatile and allow to perform elaborate manipulations. At the same time the algorithms and data structures have evolved, from the simple arrays and linked lists of the first manipulators, to the sophisticated and high performing data structures employed today. Computer architectures have evolved as well: cheap personal computers today support vectorized instructions, and they feature multi-level cache memory architectures that must be carefully exploited in order to achieve maximum performance. Not to mention the multi-core structure of modern CPUs, the widespread availability of clusters and grid architectures and the recent efforts towards the exploitation of graphics processing units for computationally-intensive tasks (GPGPU - General Purpose Computing on GPU, see Owens et al. [2007] for an overview).

Writing an algebraic manipulator that fully exploits the computer hardware is hence much more difficult today than it was decades ago.

4.1 Why the need for specialized algebraic manipulators?

A legitimate question is whether or not the need for specific algebraic manipulators still exists today. Software packages like Mathematica, Maple and even Matlab provide advanced symbolic manipulation capabilities, and computer performance is order of magnitudes higher than twenty or thirty years ago.

The answer depends on the task for which algebraic manipulations are employed. While it is true that it is certainly possible to use the aforementioned packages to perform some tasks previously the prerogative of specialized manipulators, it is at the same time true that the performance gap between specialized and non-specialized manipulators has not narrowed at all with the increase of hardware capabilities. Such performance gap is huge, and amounts to a factor of 1000 – 10000 for the most computationally intensive tasks. General manipulators, by definition, can't take advantage of prior knowledge about the mathematical entities being handled, and hence they suffer an abstraction penalty which can severely limit their performance with respect to specialised manipulators.

A performance gain of three or four order of magnitudes opens up the possibility of manipulations unthinkable with general-purpose software. As it happens in every human activity, an instrument, far from being just a mere device used to accomplish a goal, is a key element in the definition of such goal. In this sense, specific algebraic manipulators, in our opinion, will always have a prominent place in Celestial Mechanics.

The research field of algebraic manipulation for Celestial Mechanics is today very lively and active, and we had the chance to meet at international conferences various researchers actively involved with specific algebraic manipulators. All the major centres of studies on Celestial Mechanics and astrodynamics are actively working on computer algebra systems:

- in Spain the groups in Zaragoza and Barcelona working on dynamical systems develop and use extensively specific algebraic manipulators;
- the Russian school has a long tradition in the field of computer algebra systems (see, for instance, Babaev et al. [1980], Ivanova [1996, 2001]);
- in France researchers at the IMCCE (*Institut de Mécanique Céleste et de Calcul des Ephémérides*) extensively use the specialized manipulator TRIP, which has been initially written by Jacques Laskar and is now being actively expanded and enhanced by Mickaël Gastineau.

4.1.1 Why the need for another Poisson series manipulator?

Another legitimate question is why we felt the need to write yet another Poisson series manipulator, given the abundance of solutions available.

In the first place we decided to undertake this task because we needed manipulation capabilities which were not available in any other manipulator we knew of. Such capabilities were needed in the expansion of the tide-generating potential in the Earth-Moon-Sun system (see Casotto and Biscani [2004a]), a task which requires the ability to express cosines and sines of Fourier series again as Fourier series. We had tried to perform these calculation using the general system Mathematica, but quickly found out that the time needed to compute trigonometric functions of

the series of the ELP2000 theory (see Chapront-Touzé and Chapront [1988]) was in the order of nine-ten hours for each expansion. The first version of our manipulator performed the same calculation in seconds.

The research that led to this first manipulator allowed us to learn about the ubiquity of specific manipulators in Celestial Mechanics: it became clear that a Poisson series manipulator is an essential tool in this field of research. We also identified the issues that, in our view, affect the available manipulators. In particular:

1. old manipulators are tied to specific hardware or software architectures and have significant portability issues,
2. source code is generally unavailable,
3. existing manipulators are often written with specific problems in mind,
4. many of the existing manipulators use sub-optimal data structures for the representation of Poisson series,
5. computer languages commonly used (FORTRAN and C, mainly) are limiting for the genericity and reusability of existing code.

The unavailability of source code, combined with the specific nature of most existing manipulators, is in our opinion a significant drawback: in order to adapt an existing manipulator for other needs the source code should be available, since a mechanism which would allow for such an adaptation at a higher level would invalidate the speed advantage of specific manipulators over general-purpose packages. Additionally, the procedural languages, like FORTRAN and C, commonly used to develop specific manipulators, are not suited for the reuse of existing source code, because routines and data structures are often inextricably interleaved in such a manner that it is very difficult to abstract an algorithm from the rest of the code and use it somewhere else¹.

The weak points of existing Poisson series manipulators have hence become the focus of our own Poisson series manipulation framework, which is written in C++ and we call *Piranha*².

4.2 Preliminary design considerations

The considerations expressed in Chapter 2 show that the manipulation of Poisson series, being a task which can be approached from different angles, does not force particular constraints on the actual implementation of a manipulator. As it is typical with computer algebra systems, a fine point of balance between generality and performance must be found. The utopian goal is that of conceiving a program that, while being effective in fulfilling the purpose it was written for in the first place, can also be extended to a broader range of action in the future.

This is particularly true when dealing with Poisson series. As we saw in the last chapter, for instance, there is a fundamental difference between the computer representations of generic Poisson series and Fourier series (see §2.1.4): it is clear that, for the representation of the coefficients,

¹This reason alone may explain the abundance of Poisson series manipulators.

²The name is a pun based on the fact that “poisson”, in French, means “fish”.

4 Designing a modern Poisson series manipulator

when dealing with Fourier series it will be sufficient to use one of the basic datatypes available in every computer language (be it Fortran's `REAL` or C/C++'s `double`). If we want to be able to manipulate Poisson series with polynomial coefficients, instead, we will need to introduce a new datatype for the representation of polynomials and a set of functions to manipulate it. Additionally, we may desire to be able to use non-standard datatypes also when dealing with Fourier series: for example it may be useful for some applications to be able to represent the numerical coefficients as rational numbers. It is evident that in these three cases there will be differences in the manipulation of the coefficients, but the functions used for the manipulation of the trigonometric parts and for their interaction with the coefficients will not have to be changed.

In other words, if we want to achieve maximum code reusability and genericity, the program should be written *once* and for *all* the different datatypes on which it is going to operate. In computer science this datatype-independent paradigm of programming is known as *generic programming*.

4.2.1 Generic programming

Generic programming (see Musser and Stepanov [1989], Dehnert and Stepanov [2000], Dos Reis and Järvi [2005]) is a programming paradigm that focuses on finding commonality among similar implementations of the same algorithm, then providing suitable abstractions in the form of concepts so that a single, generic algorithm can realize many concrete implementations.

A simple example of generic programming in C++ can be seen in the following listing:

```
1 template <typename T>
2   T max(T x, T y)
3   {
4     if (x < y)
5       return y;
6     else
7       return x;
8   }
```

Here a maximum value function is defined. The input datatype is *not* specified, instead it is declared through the `template` statement that the function will operate on a generic type which will be known as `T`. When the function is used to find the maximum of, e.g., two integers, the compiler will substitute `T` for `int` during compilation. In other words, for all the datatypes for which the `<` operator is defined, this function can be used in a completely generic way. In C++ generic functions are known as *template functions*.

Another popular example of generic programming is that of a sorting algorithm that can operate on different data structures, like arrays and linked lists.

Generic programming is a feature which is found, under various denominations, in most modern computer languages. An incomplete list includes ADA, BETA, C++, Objective-C, D, Eiffel, Java, ML, C# 2.0, Chrome 1.5, Visual Basic .NET 2005 and Haskell. Particularly with reference to the C++ language, where it goes by the name *template programming*, generic programming is today a very active field of research in the computer science community (see Vandevoorde and Josuttis [2002] and Alexandrescu [2001]). A notable example of the power of template program-

ming can be seen in Veldhuizen [1995], where it is shown how a technique known as *expression templates* is used to mimic the behaviour of FORTRAN arrays without introducing extensions to the core language (see also Haney [1996]).

4.2.2 Choosing a computer language

Beside the availability of generic programming facilities, we chose to develop Piranha in C++ for a number of other reasons, which are summarised below:

- C++ is a Object-Oriented (OO) language: although not all of the usual features of an OO language are used inside Piranha, some concepts are extremely useful in the context of a software project which aims to be extensible, modular and generic. A detailed list and a brief explanation of C++'s features used in the manipulator are given in §4.3.
- Performance: C++ performance edge with respect to other OO programming languages is twofold. First of all C++ is compiled to native code (unlike Python or Ruby – which are interpreted languages). Native code delivers top performance, and as we saw earlier performance is one of the primary concerns in a specific manipulator. Secondly, just like its forefather C, C++ allows a very fine-grained control over the resources used by the program, particularly (but not only) regarding memory management. Where other languages offer some kind of automatic memory management (or *garbage collection*) in C++ the programmer has to take care of allocating/deallocating memory areas. While this can be cumbersome (but it can be made almost transparently in C++ – and there's always the possibility to use one of the many existing memory managers written in C++), on the other hand it enables focused and efficient memory management strategies.
- Operator overloading: this features allows to redefine the meaning of mathematical operators in source code. I.e., given two series A and B it is possible to write $A+B$ instead of $A.add(B)$. This leads to compact and readable code, especially in scientific software. Operator overloading is a notable feature missing in other languages (it's not present in Java and C, for instance).
- Popularity: C++ has been introduced more than two decades ago and today it is one of the most popular languages. It has been used to write every kind of software, from operating systems to office suites, from games to advanced scientific software packages. As result there is an astounding number of high quality libraries written for many different purposes: networking, database access, graphics, GUI (Graphics User Interface) programming, etc. A software written in C++ can be extended at a later date in every unthought direction. Additionally, for the same reason there are lots of high quality programming tools for C++, like debuggers, profilers, code documentation tools, etc.
- The Standard Template Library (STL): the STL is a software library included in the C++ standard, providing many useful programming facilities like containers (vectors, linked lists, sets, maps), algorithms, iterators and functors. The STL was created as the first library of generic algorithms and data structures, with two main ideas in mind: generic programming and abstractness without loss of efficiency (see Stepanov [2007]).

4 Designing a modern Poisson series manipulator

- The Technical Report 1 (TR1): TR1 is a draft document specifying additions to the C++ Standard Library such as regular expressions, smart pointers, hash tables, and random number generators (see Austern [2005]). TR1 is not yet standardized, but it will likely be part of the next official standard mostly as it stands now. The GCC compiler suite already implements many parts of TR1.
- Portability: C++ compilers are available for almost any hardware platform, hence a C++ program that is written in a portable way (i.e., without using architecture-specific features) will run without modifications almost everywhere. This is particularly true when using the GNU C++ compiler, which (as of version 4.x) supports more than 20 different hardware platforms.

To summarise, our choice was dictated mainly by the fact that C++ is the highest-level language that does not compromise on pure performance³. Particularly, C++'s compile-time genericity (as opposed to the powerful runtime dynamicity available, for instance, in Objective-C) is in our opinion C++'s major selling point in the context of high-performance scientific software with respect to other OO languages.

4.2.3 The three lives of Piranha

The current state of the Piranha is the result of a trial and error process, partly prompted by the inevitable clash between early theoretical design decisions which were challenged by on-the-field results and partly because of the author's own need to get acquainted with a new language (C++) and with the development of a medium-sized project like this one. The history of Piranha consists of three phases:

1. initially Piranha was conceived as a set of routines in plain C. Memory management was done through manual `malloc()` and `free()` calls. Coefficients and trigonometric multipliers were stored in large separate arrays, and the pointers to these arrays were stored inside data structures (`struct`, in C) which represented the Poisson series. Operations on series were implemented as functions (e.g., `psadd()`) and the series could only have numerical coefficients (i.e., only the manipulation of Fourier series was implemented).
2. The second phase saw the transition to an object-oriented paradigm. The `struct` construct was replaced by the `class` construct, and operations on series were implemented as methods. Apart from the change to OO programming, the internal algorithms remained the same. Coefficients and trigonometric multipliers were still stored in manually managed chunks of memory (but this time C++'s `new` and `delete` operators were used instead of the `malloc()` and `free()` functions). Operator overloading was introduced to perform operations among series.

³As a side note, we have often heard the remark that C is faster than C++ because it works at lower level. We frankly find this statement rather strange, since, minor differences aside, C++ is a superset of C, and anything that can be done in C can also be done in C++. There is really no reason because of which the choice of C++ over C would inherently lead to performance penalties.

3. The third rewrite of Piranha was a major enhancement which saw a radical change in the representation of the series, the use of scoping for memory management, the employment of more advanced data structures (instead of the plain arrays used previously) and the focus on generic programming. Such a huge change was made possible largely thanks to the adoption of a set of high-quality C++ libraries called *Boost* (see §4.4 for an overview). As a result Piranha is now much lighter in terms of lines of code than it was before, it offers a much more intuitive way of interacting with the series and can be extended to manipulate new types of Poisson series in a very compact and easy way, without touching the core source code.

We believe that the current design of Piranha, while certainly not perfect and susceptible to improvements, establishes solid foundations, and enables future extensions in a seamless and coherent way. It is expected that future work on Piranha will consist mostly of feature additions and optimization work.

Currently Piranha compiles, runs and has been tested on GNU/Linux with GCC (GNU Compiler Collection) versions 3.4.x and 4.x, on FreeBSD with GCC 3.4, and on Windows XP using the MinGW port of GCC (currently version 3.4). It should run without modifications on all Unixes where at least GCC 3.4 is available.

4.3 C++'s features used in Piranha

Piranha takes advantage of most of C++'s features. C++ is a feature loaded language, to the point of resulting somehow confusing at times. There is often more than one way of doing things in C++, and hence particular care must be taken in the planning phase to minimize the effort by choosing the most efficient way to solve a problem.

C++'s feature set is made of all the typical OO features, plus some welcome additions (like operator overloading). In the following paragraphs we will recall some programming concepts used in Piranha. For a more complete description of C++'s capabilities we suggest to refer to Eckel [1995].

4.3.1 Namespaces

Namespaces are a relatively new addition to C++'s arsenal, and are best described as context identifiers. Simply put, namespaces are abstract domains to which classes and functions are associated. The main use of namespaces is that of being able to define functions (or classes) which in that particular namespace have some meaning, while in another namespace they have a different meaning. The purpose of associating objects to a domain is hence twofold:

1. to define functions with common and meaningful names (e.g., `print()`, `load()`, etc.) avoiding name clashes with other libraries, and
2. to group logically related functions under a single identifier.

An example is probably the best way to explain namespaces:

4 Designing a modern Poisson series manipulator

```
1 namespace 3rd_party_library
2 {
3     void do_nothing();
4 }
5
6 namespace my_library
7 {
8     void do_nothing();
9 }
10
11 int main()
12 {
13     3rd_party_library::do_nothing();
14     my_library::do_nothing();
15     return 0;
16 }
```

Here we can see two namespaces, `3rd_party_library` and `my_library`, each one containing the declaration of a `do_nothing()` function. Because each function is positioned inside a different namespace there are no conflicts, even if the functions share the same name. The choice of which function to use is made when the namespace specification (e.g., `my_library::`) is prefixed to the function name. Namespaces feature also the possibility to use as default certain functions from one namespace and other functions from other namespaces.

Placing a library's functions and classes inside a namespace is good practice, since it prevents potential conflicts with names in other libraries. In Piranha namespaces are used mainly with organizational purposes, i.e. they are used to group all the functions and the classes used by the manipulator. Piranha's root namespace is named, unsurprisingly, `piranha`.

4.3.2 Classes

Classes are the single most important entity in OO programming, and they can be described as advanced containers of heterogeneous data. Like a C `struct` a C++ `class` groups together (*combines*, in computer science language) different data types into a composite one. Additionally, a class can also:

- embed the functions (*methods*) used to communicate with its data,
- provide a public and a private interface to the data,
- relate to other classes in a parental relationship.

The third property means that classes can have children classes, i.e. classes that inherit their parent's properties but also add their own data and methods. A canonical example is that of the class `Vehicle` and its child (or *subclass*) `Car`. `Vehicle` has a `fuel` property and a `start()` method, which `Car` will inherit. Additionally `Car` also provides the `wheels` and `brakes` properties, and the `honk()` method. `Car` is a `Vehicle`, and it is *more than a* `Vehicle`: the act

of deriving a class from another one is called *inheritance*. Inheritance promotes code reuse, since for the subclasses the inherited methods won't have to be rewritten. Additionally, it encourages a structured thinking (and programming) model, in which objects are ordered hierarchically.

Classes accomplish two of the main goals of OO programming:

- modularity: classes are independent units that act on each other. They have an autonomous life, and can be re-used in other contexts;
- encapsulation: information on how the class works can be concealed inside the so-called *private* methods of the class. A user of the class can be exposed to a *public* interface, the unneeded implementation details being hidden. For example, a `Car`'s user knows that a `Car` can `honk()`, but she won't care about the (private) implementation details of that method.

Classes are naturally fit to be used in scientific and mathematical problems, since they mimic the kind of mutual interaction that is typical of mathematical entities. Each object has a set of methods to interact with other objects, and each object groups methods and data in a single package. In Piranha Poisson series are represented as classes, and they are manipulated through methods.

Piranha does not make use of some other properties of classes, such as abstraction and dynamic polymorphism. While these concepts are very appealing from a programming point of view, they introduce performance penalties that can become too much cumbersome for a computationally-intensive software. Instead Piranha uses a template programming design pattern known as *Curiously Recurring Template Pattern*, or CRTP (see Coplien [1995]), which allows to achieve a form of static polymorphism which does not incur in runtime performance penalties.

4.3.3 Template classes

In §4.2 we briefly introduced the concept of template function, i.e., a function that is written to operate on generic datatypes instead of specific ones. In C++ this concept is extended to classes.

Let us suppose, for instance, to write an implementation of a linked list of integers in C++. The linked list class will be called `List_int`. If, in a second stage, we need a linked list of real numbers, without generic programming facilities we will probably copy and paste the implementation of `List_int` and substitute all the occurrences of the `int` keyword with `double`. This strategy however is far from optimal, since:

- manual copy and paste is error-prone and
- if a bug is fixed in the original class it will be needed that all the copied classes receive the same fix (which can become quite intricate and error-prone too).

C++'s solution to this problem is to provide a template class, i.e. a class whose datatypes are not specified until compile time:

```
1 template <class T>
2   class List
```

4 Designing a modern Poisson series manipulator

```
3 {
4   implementation details;
5 }
6
7 int main()
8 {
9   List<int> int_list;
10  List<double> real_list;
11  return 0;
12 }
```

Here we can see a conceptual example of a `List` template class. `List` is declared as a regular class, except for the single line `template <class T>` prefixed to the declaration. This line specifies that the following class declaration will use an unspecified datatype which will be called `T`. The methods of the class will be in turn template functions, i.e., they will operate on the generic datatype `T`. In the main body of the program two objects, `int_list` and `real_list`, are created: they are two different instantiations of the same template class `List`, one instantiated for integer numbers, the other for real numbers. Note that the `T` type is not limited to built-in datatypes, it can be any valid C++ class. Additionally, implementations of the `List` class specialised for specific types (e.g., strings) can be used to override the default template implementation in a procedure known as *template specialisation*.

The advantage of using templates is clear especially when dealing with classes whose behaviour is datatype-independent. In the case of linked lists (and of data containers in general) generic programming is a particularly appropriate programming paradigm. What matters is the interface with the container, hence the methods used to insert and erase elements, which will not vary according to the content. *What* is contained is of secondary importance, and the dull job of generating appropriate code for each version of the template class is delegated to the compiler.

In Piranha template classes are used to design Poisson series whose coefficients are arbitrary entities: they can be real numbers, arbitrary precision numbers, polynomials, arbitrary algebraic expressions, etc. The only prerequisite is that the coefficients' types support a set of properties that define a concept (see §5.1).

4.3.4 Operator overloading

Operator overloading, strictly speaking, does not introduce new features to C++, and it is probably best described as *syntactic sugar*. Operator overloading simply allows to use familiar mathematical (but not only) operators instead of explicit calls to methods. Let's assume for instance that it is needed to add two Poisson series, `P1` and `P2`. Without operator overloading the code is something like

```
P2.add_to(P1)
```

With operator overloading it is possible to wrap the `add_to()` method inside the `+` operator, and simply write

```
P1+P2
```

Nothing has changed, the `add_to()` method is still called to perform the actual operation. However the notation in the source code is certainly more familiar. Operator overloading hence leads to a much more compact and readable code, easing both the identification of software bugs and the acquaintance with the source code of an external developer. Operator overloading is used extensively in Piranha.

4.3.5 Iterators

Iterators can be seen as a generalisation of C pointers. They are objects which allow a programmer to traverse through all the elements of a collection, regardless of its specific implementation. Iterators, just like C pointers, can be dereferenced through the `*` operator, and they can be incremented through the `++` operator. An iterator for an array-like container, for instance, is simply a pointer, and the `++` operator simply increases the pointer value using pointer arithmetics. An iterator for a linked list, instead, will overload the `++` operator so that it will wrap the linked list method to reach the next node of the list.

The combination of iterators and operator overloading allows to write generic algorithms that will work regardless of whether the underlying data structure is an array-like or a list-like container⁴.

4.4 The Boost libraries

Boost (<http://www.boost.org>) is a set of C++ libraries created by a community of developers led by some of the members of the C++ Standards Committee. From the website homepage:

“Boost provides free peer-reviewed portable C++ source libraries.

We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use.

We aim to establish “existing practice” and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee’s Library Technical Report (TR1) as a step toward becoming *part of a future C++ Standard*. More Boost libraries are proposed for the upcoming TR2.”

(emphasis added). Boost is highly regarded in the free-software programming community, and it has been adopted by many commercial software developers as well. The Boost libraries are used, among others, by Adobe (Photoshop CS2 and Indesign), Real, McAfee, DataSolid GmbH Germany (CADdy++ Mechanical Design) and Dimesion 5 (Miner3D).

Initially the introduction of Boost in Piranha was prompted by a specific feature, the multi-index container which will be introduced shortly (see §4.5.3). However soon it became clear that Boost could provide many other interesting features, and as time passed Piranha began to

⁴Of course this feature will not exempt the developer from being aware of the different performance characteristics of arrays and linked lists.

4 Designing a modern Poisson series manipulator

rely on other facilities provided by Boost (e.g., string handling, n-tuples, iterators management, multi-threaded programming, etc.).

In order to ensure efficiency and flexibility, Boost makes extensive use of templates and template programming. Boost is a source of extensive work and research into generic programming and metaprogramming in C++.

4.5 Data structures for Poisson series

Piranha's design concepts stem from the analysis of Poisson series manipulation developed in Chapter 2. In this section the fundamental conclusions of that analysis will be recalled and we will show which programming devices, in our opinion, are best suited to the software modelling of a Poisson series. Such software constructs, described briefly in the following subsections, are analyzed thoroughly in Knuth [1998b] and Cormen et al. [1990].

4.5.1 Ordering of terms: binary search trees

As we have seen in §2.3.1 it is necessary to introduce an ordering in a Poisson series. This ordering is used in multiplications, when it is typically needed to truncate the series to limit the quadratic growth of the number of terms.

In the previous incarnations of Piranha, series were ordered only when needed (i.e., only during multiplications), whereas during addition/subtraction terms were inserted at the end of the series in a sequential fashion. This strategy had two disadvantages:

- series' ordering was generally undefined,
- each time an ordering was required, a full reordering of the series was needed.

Multiplications are very frequent operations, and form the basis of more advanced manipulations. Sorting algorithms typically perform in $O(n \log n)$ time, hence it is clear that frequent complete sortings can become a bottleneck when dealing with many multiplications and long series.

The current version of Piranha improves the situation with the introduction of *self-balancing binary search trees*. A self-balancing binary search tree (SBBST) is a special case of a data structure known as *binary search tree* (BST), which in turn is a kind of *binary tree*.

Definition A *binary tree* is a tree data structure in which each node has at most two children.

Typically the child nodes are called *left* and *right*.

Definition A *binary search tree* is a binary tree which has the following properties:

- Each node has a value.
- A total order is defined on these values.
- The left subtree of a node contains only values less than the node's value.
- The right subtree of a node contains only values greater than or equal to the node's value.

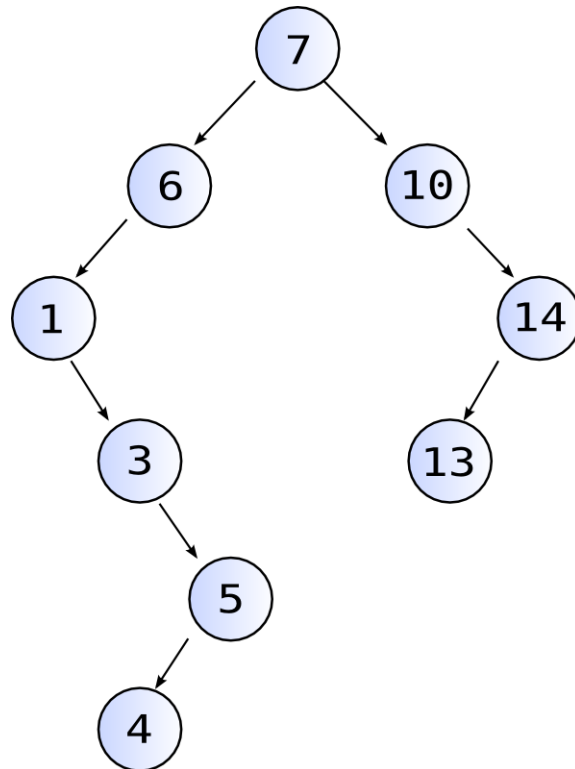


Figure 4.1: Binary search tree (unbalanced).

Definition A *self-balancing binary search tree* is a binary search tree that attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible at all times, automatically.

A simple example of an unbalanced BST can be seen in Figure 4.1. The same binary search tree is displayed in Figure 4.2 after being height-balanced.

An important property of BSTs is that they keep the data ordered: a new node is added to the tree by *traversing* (or *walking*) the tree node by node until a suitable position is found. If the new element is greater than or equal to the current node then it is directed towards the next node in the right subtree, otherwise it is directed to the next node in the left subtree. If there are no next nodes, the new element is appended where the next node should be.

The advantage of SBBSTs over plain BSTs is that insertion, look-up and removal of elements are all performed in $O(\log n)$ time, where n is the number of nodes of the tree, and that this complexity is guaranteed for worst cases. Unbalanced BSTs, instead, have average $O(\log n)$ complexity, but in the worst case (when all nodes have only left – or right – children) the BST resembles an ordered linked list, and as such has $O(n)$ complexity. If BSTs are not kept balanced they quickly degrade into ordered linked list-like trees as nodes get added. In SBBSTs, hence, once a series is ordered it is relatively cheap to insert new elements in subsequent manipulations. SBBSTs are kept balanced by algorithms that at key times re-arrange the tree so that its height

4 Designing a modern Poisson series manipulator

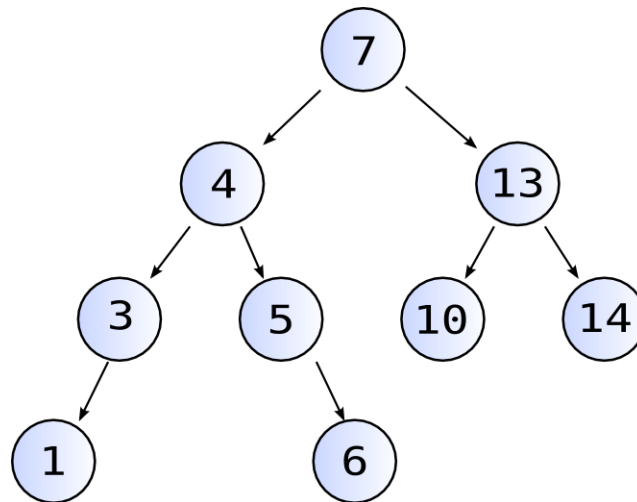


Figure 4.2: Balanced binary search tree.

is minimized.

In Piranha series terms are stored as nodes of a SBBST in which the ordering is established by a property of the terms' coefficients. In the case of Fourier series, for instance, ordering is based upon the coefficients' absolute value. Multiplication begins from the leading term of the series, and the binary tree is traversed in descending order until the end of the series (or the truncation level) is reached. This kind of *tree traversal* of a BST is called in-order tree traversal, and it is just one of the ways a tree can be explored. Without going into much detail, we'll just mention that in-order tree traversal is a recursive algorithm which starts from the root node (i.e., the topmost node) and consists of these steps:

1. for the current node check whether it has a left child. If that's the case then go to step 2 or else step 3.
2. Repeat step 1 for this left child.
3. **The next node is found.**
4. For the current node check whether it has a right child. If that's the case then go to step 5.
5. Repeat step 1 for this right child.

By applying this algorithm to the tree in Figure 4.2 at each iteration we find the following sequences:

- 7-4-3-1
- 3
- 4

- 5
- 6
- 5-4-7
- 13-10
- 13
- 14

Note how the last numbers of each iteration form an ascending sequence. This sequence can be reversed by simply changing the ordering criterion of the tree or by traversing right-to-left instead of left-to-right.

In C++ SBBSTs are usually implemented as template classes, so that the nodes can contain any type of data, and they offer a convenient interface that hides the intricacies of their management. In-order traversal, for example, is implemented using iterators. In the following listing we can see a SBBST class `Bst` which provides a `begin()` method:

```

1 template <class T>
2   class Bst
3   {
4     implementation details;
5   }
6
7   int main()
8   {
9     Bst<int> bst_int;
10    iterator iter=bst_int.begin();
11    // iter is augmented by one
12    ++iter;
13    // iter now points to the second
14    // biggest element of bst_int,
15    // and it can be used to retrieve
16    // the information stored in
17    // that node.
18    return 0;
19  }
```

The `begin()` method returns an iterator to the top value of the tree. The `++` operator has been overloaded for iterator objects to implement the in-order traversal to the next node in a way that is transparent to the programmer.

Template implementation of SBBSTs are available in the STL (the `set` and `map` classes).

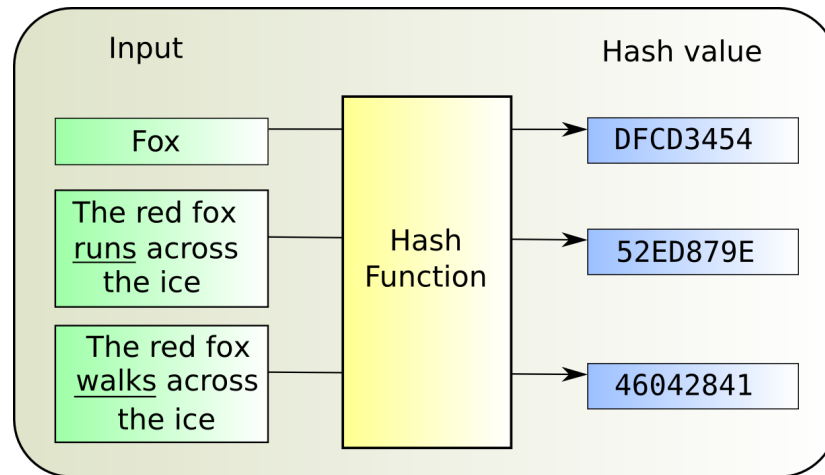


Figure 4.3: A hash function operation on strings and outputting hexadecimal values.

4.5.2 Identification of terms: hash tables

As we saw in §2.3 the fundamental operation performed on Poisson series is the insertion (or deletion) of a term. We also introduced the concept of term packing, and stressed the importance of being able to identify existing terms in Poisson series in order to preserve the canonical form.

A possible way to identify a term in a Poisson series is to use a BST: terms are nodes of the tree, and the ordering criterion is lexicographic on the elements of the i -vectors. A binary search on the tree can then be performed in $O(w \log n)$ time, where n is the length of the series and w its trigonometric width.

A more efficient solution is to employ a *hash table* (also known a *hash map* or *dictionary*). Hash tables are unsorted containers which use a particular function (called *hash function*) to establish a mapping between an object and its position in an array-like container. This function provides a way of creating a small digital fingerprint (called *hash value*) from the data (see Figure 4.3). The hash value is commonly represented as a short string of binary data but for the use in hash tables it is typically converted into an unsigned integer value through a *range-hashing function* to determine the position (or the *index*) of the data in an array-like structure (see Figure 4.4).

This way it is possible to establish if an object is present in a data set simply by computing its hash value, go to the corresponding position in the array and see whether the slot is taken or not. The hash value however typically cannot be unique (since in most circumstances one is interested in having a fingerprint significantly smaller that the object it maps to) and hence there will be *collisions* in the hash tables (i.e., slots in the hash tables that will contain more than one element). A common solution is to have singly-linked list in every slot that will store the elements that map to that slot (an approach called *separate chaining*). An ideal hash function minimizes the need for collision resolution by randomizing the mappings to slots. Once many slots are taken (i.e., the *load factor* of the hash table grows near one) the container is usually resized, so that slots are re-assigned in a randomized fashion and collisions decrease in number.

Technicalities aside, we can assume in this context that finding an object in a hash table involves

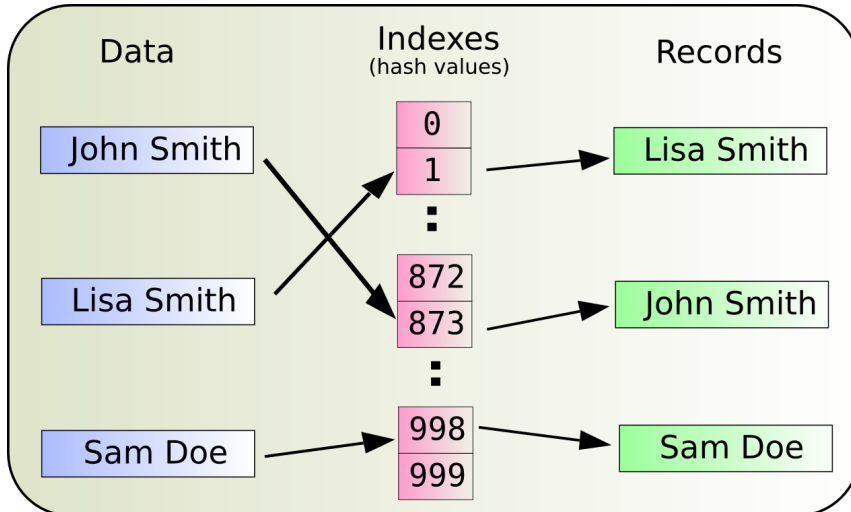


Figure 4.4: A hash table storing strings.

a constant number of mathematical operations, and as such the operation is performed in average $O(1)$ time. This is a marked improvement with respect to the logarithmic complexity of the BST-based solution. In the case of Poisson series, the hash value is computed from the i -vectors, which can be thought of as fixed-size strings. We are not interested in the hash values of coefficients, since we need to identify terms only according to their trigonometric parts.

Writing a good hash table can be tricky, and the following issues can lead to disastrous performance drops:

- poor hash function,
- poor strategy of collision resolution,
- poor range-hashing function.

The multiplicative hashing function advocated in Knuth [1998b], for instance, has notoriously poor clustering behaviour.

Nevertheless, hash tables are widely used in diverse fields of information technology, and, when correctly implemented, offer the best performance as far as unordered containers are concerned (see, for instance, the analysis in Heinz et al. [2002]). We discuss in more detail the characteristics of the hash function used in Piranha in §5.6.2.

4.5.3 The `boost::multi_index_container` class

In §4.5.1 and §4.5.2 we have identified two data structures that efficiently fulfill the requirements for the manipulation of Poisson series. SBBSTs and hash tables should be combined in a container which provides a twofold access semantic: we will need a hash table for fast term identification, and a SBBST to keep the series ordered and apply a truncation methodology during series multiplication.

4 Designing a modern Poisson series manipulator

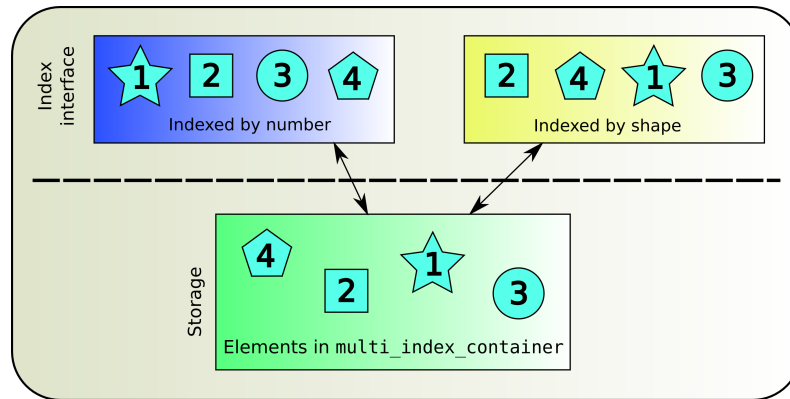


Figure 4.5: Diagram of a `boost::multi_index_container` with two indices.

Initially we had implemented this double container class by hand, using a combination of the ordered and hashed containers available in the STL and in the TR1. The current implementation of Piranha uses, instead, the more flexible and powerful solution provided by the Boost class `multi_index_container`. This class enables the construction of containers maintaining one or more indices with different sorting and access semantics. This means that:

1. it is possible to sort generic objects (and keep them sorted) in more than one way and with respect to more than one of their properties. This concept is displayed graphically in Figure 4.5;
2. it is possible to access the data in the container using different algorithms (at the moment binary search, hashed search, unsorted linked list traversal and random access are implemented).

`boost::multi_index_container` is based around the concept of *index*. An index is an interface that provides a viewport on the data stored in the container. As such the index specifies:

1. the ordering (i.e., whether the viewport is sorted or not and according to which properties of the data the sorting takes place) and
2. the access semantics of the viewport (i.e., *how* the data is accessed).

As an example let's suppose that we want to create a list of `employee` objects, each one containing a name and an ID field. We want to be able to sort employees with respect to their names and their IDs. This can be done through the following instantiation of a `multi_index_container` (which has been split on several lines to increase readability):

```
1 typedef multi_index_container<
2   employee,
3   indexed_by<
4     ordered_unique<identity<employee> >,
```

```

5     ordered_non_unique<member<employee, std::string, &employee::
        name> >
6     >
7     >
8     employee_set;

```

Let's take a look line by line:

- on the first line the `typedef` directive specifies an alias for the complicated datatype that follows, which will be called `employee_set` (as specified on the last line). This is a *type definition* and it is used to give short familiar names to complicated object types used in the source code;
- on the second line we specify that the containers will store objects of type `employee`;
- on the third line we declare that the container is indexed in two different ways:
 1. the first (line 4) is a unique sorting in which the sorting criterion is the “<” operator⁵, which has somewhere been overloaded for the `employee` object to mean that an `employee` is *less than* another one if its ID is smaller. The `identity` function, in other words, means that the order is decided by the default function for comparisons, which is the “<” operator. This sorting is unique because there cannot be two `employees` with the same ID.
 2. The second (line 5) is a non-unique sorting based on the `name` member of the `employee` object, which is of type `std::string` (the standard string definition of C++). The `&` syntax means that we are passing the pointer to `employee`'s `name` member instead of the member itself and is just a C++ technicality. The ordering is established by the action of operator “<” on strings, which in C++ means alphabetical order. This sorting is not unique because two `employees` can have the same name.

Please note that the declaration of this `multi_index_container` involves template functions and template classes used as parameters of other template functions and classes. Template programming in C++ can indeed become quite complicated.

Now the `employee_set` class we just defined can be accessed in two different ways. We can declare a viewport on the ID index (the first one defined), and access it like this (assuming that an `employee_set` called `es` has been instantiated and filled with elements previously):

```

1     const employee_set::nth_index<0>::type& ID_index=es.get<0>();
2     employee first_employee>(*ID_index.begin());
3     first_employee.print_name();

```

On the first line a viewport called `ID_index` is defined. The `const` keyword and the `&` symbol declare that this kind of object is a constant reference, which is reached by the `get()` method of `employee_set`. This method is called with the integer parameter `0` meaning that we are requesting a viewport on the first index (in C/C++ the first element of a collection is given

⁵The “<” operator is not to be confused with the angle brackets “<>”, which, as seen in §4.3.3, are used in the declaration of templates. In these listings “<” is never used as an operator.

4 Designing a modern Poisson series manipulator

0 as index). On the second line an `employee` object is defined by copying the first employee of our viewport (reached through the `begin()` method – the `*` symbol being another C++ technicality). This means that `first_employee` is the employee with the lowest ID (since it is placed at the beginning of the ID index). Finally on line 3 employee's `print_name()` method is used to display `first_employee`'s name.

By simply changing `get<0>` with `get<1>` it is possible to access the other index, the one based on the employees' names:

```
1 const employee_set::nth_index<1>::type& name_index=es.get<1>();
2 employee first_employee>(*name_index.begin());
3 first_employee.print_ID();
```

This time `first_employee` is the first employee in alphabetical order, and on line 3 its ID gets printed.

Beside the possibility of multiple sorting, of yet higher importance for Piranha is the ability of having more than one access semantics to the data. Let's take a look at this slightly modified `employee_set` declaration:

```
1 typedef multi_index_container<
2     employee,
3     indexed_by<
4         hashed_unique<member<employee,int,&employee::id_num> >,
5         ordered_non_unique<member<employee,std::string,&employee::
6             name> >
7 > > employee_set;
```

The only difference with respect to the previous declaration is on line 4, where the ID sorted index is replaced with a hashed index. This index requires the definition of a functor that computes the hash value of the integer `id_num` member of the `employee` class, which must be provided by the developer. Now the hashed viewport honours the access semantics of unsorted hashed containers, and as such it features $O(1)$ search time performance. For instance:

```
1 const employee_set::nth_index<0>::type& ID_index=es.get<0>();
2 employee first_employee(*ID_index.find(1));
```

Here we use the hashed viewport `ID_index` to search for the employee whose ID is 1 in $O(1)$ time. If the `ID_index` viewport had been defined as a sorted index this same search would have been performed in $O(\log n)$ time instead.

`boost::multi_index_container` solves the recurrent need for different access interfaces to a container, and as such is a testimony of C++'s power and expressiveness. While usual data containers (arrays, sets, maps, hashed containers) are commonly found in most programming languages (either built-in or provided by external libraries – in C++ they are provided by the STL), to our knowledge C++ is the only one which provides such a powerful construct.

In Piranha `boost::multi_index_container` is used as a container for the terms of a Poisson series. Two indices are defined by default, one ordered according to the coefficients, the other one being a hashed index on the terms' i -vectors. As seen in the above sample declarations

it is very easy to add new indices and access semantics. When dealing with perturbing functions, for instance, it may be useful, with the task of term classification in mind, to have an index ordered according to the frequencies of the trigonometric parts. To achieve this it is enough to add a single line of code to the series definition and a function of few lines to implement the comparison of frequencies. Another possibility is the use of an array-like random access index to be used when parallelizing the code (most parallelization techniques require random access to the elements of a container).

4 Designing a modern Poisson series manipulator

5 PIRANHA: ARCHITECTURE AND IMPLEMENTATION DETAILS

Experience is what you get when you were expecting something else.

– *Unknown*

IN Chapter 4 we introduced the basic ideas upon which our Poisson series manipulator, Piranha, is built. In this chapter we will discuss in more detail the implementation of such ideas. It is not our intention to cover all Piranha’s source code; instead we will focus on the most important aspect of the architectural design, in order to provide an overview of its most relevant features. We will also discuss in deeper detail how certain operations, critical for the efficiency and speed of a Poisson series manipulator, are performed.

5.1 Main classes for Poisson series

In Figure 5.1 the classes used in Piranha to represent Poisson series are shown. Starting from the bottom, we can see that the base series class contains a collection of instantiations of term classes (i.e., a series is a *container* of terms). The term class, in turn, is a *composition* of a coefficient class and a trigonometric part class.

We refer to a *base* series class because such class is not meant for direct use. The base series class, instead, implements a small set of primitive manipulation methods that will be used to compose higher level operations (i.e., those operations which are meant to be employed by the end user) in a *derived* series class. This way the base series class will not be littered with specialised methods which will be instead implemented in the derived classes. We see this approach as an implementation of the schema envisioned in Henrard [1988], where a separation between base and specialised manipulators is hoped for and advocated.

The term class is a simple container for the pair formed by coefficient and trigonometric part. The term class is a template class, which means (as we recall from §4.3.3) that the class types representing the coefficient and the trigonometric part are not fixed, but they are decided at compile-time. Obviously, not any type can be used as a coefficient or as a trigonometric part; inside the base series class and the term class, indeed, certain methods from the coefficient and trigonometric class will be called: if such methods are not present, a compile error will be emitted¹. In the jargon of the modern C++ programming paradigm, it is said that the base series class and the term class implicitly define a *concept* (see Gregor et al. [2006]) for the representation of coefficients and trigonometric parts.

¹This is analogue to the template function example shown in §4.2.1, where it is requested that the generic type T supports the “<” operator.

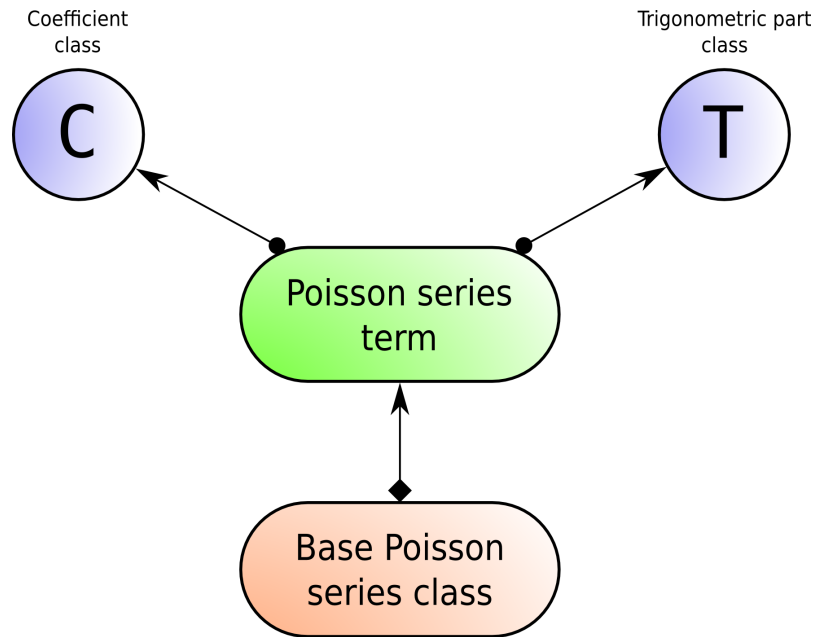


Figure 5.1: Piranha’s hierarchy of fundamental classes for the representation of Poisson series.

Definition In the context of C++ generic programming, we refer to *concept* as a set of properties (methods, operators, data members, typedefs, etc.) that a class must expose in order to be used in other specified generic classes or functions.

Definition In the context of C++ generic programming, a *model* is defined as a class that fulfills the requirements of a specific concept.

It is important to stress that the adherence of a class to a concept is a compile-time check which does not incur in any overhead at runtime.

5.1.1 Example: basic Poisson series coefficient concept

As an illustrative example, we report here some of the methods required by the concept defining the coefficient of a Poisson series. In order to be used as a coefficient in the base series class, a class must implement, among others,

- constructors from C++’s numerical “plain old types” (i.e., `int` and `double`),
- a constructor from a standard C++ string (to be used when reading series from file),
- an empty constructor which will initialise the coefficient to zero,
- a copy constructor,
- printing methods for output to screen or file,

- a `norm()` method which returns the absolute value of the coefficient,
- an `evaluation()` method for the numerical evaluation of the coefficient at a certain time,
- a `swap()` method for swapping content with another coefficient,
- an `invert_sign()` method,
- `add()` and `subtract()` methods,
- a `divide_by()` method,

and so on. The definition of such a concept comes both from mathematical necessities (e.g., the coefficient must know how to divide by two in order to implement the basic Werner trigonometric formulas) and from efficiency considerations (e.g., a `swap()` method is not really necessary for Fourier series, where the coefficients are lightweight numerical types and copying them around is cheap, but it can considerably improve performance when dealing with polynomials if implemented through direct pointer manipulation).

More advanced coefficient concepts can be defined. The application of the Jacobi-Anger expansion for the calculation of circular functions of Poisson series, for instance, requires the capability to calculate Bessel functions of the first kind of the coefficients (see §3.2). We can then define a trigonometric coefficient concept that inherits all the characteristics of the basic coefficient concept and adds a `besselJ()` member for the calculation of J_n .

Defining a concept is mainly a matter of craftsmanship, and requires a great deal of actual work on the source code. Great care and attention should be put into this task. The rewards of such a work are the rigorous definition of the software architecture and the minimization of the effort needed to create new models. In our experience with Piranha, the formal specification of concepts (which is a work in progress since not all the relevant classes of the framework explicitly adhere to a standardized concept yet), being intimately entwined with the coding process, has been one of the major, most time-consuming tasks.

5.2 Anatomy of the base series class

As we hinted near the end Chapter 4, the core of the base class used to represent Poisson series in Piranha is a `boost::multi_index_container` class (see §4.5.3) which acts as a container for terms. As we already explained, two indices are present by default:

1. a sorted index which keeps the terms ordered according to some property of the coefficients,
2. a hashed index in which the hash value is computed from the trigonometric multipliers.

The sorting criterion depends on the characteristics of the coefficient class, and more indices can be added, since the index specifier is a template parameter of the base series class. It could be

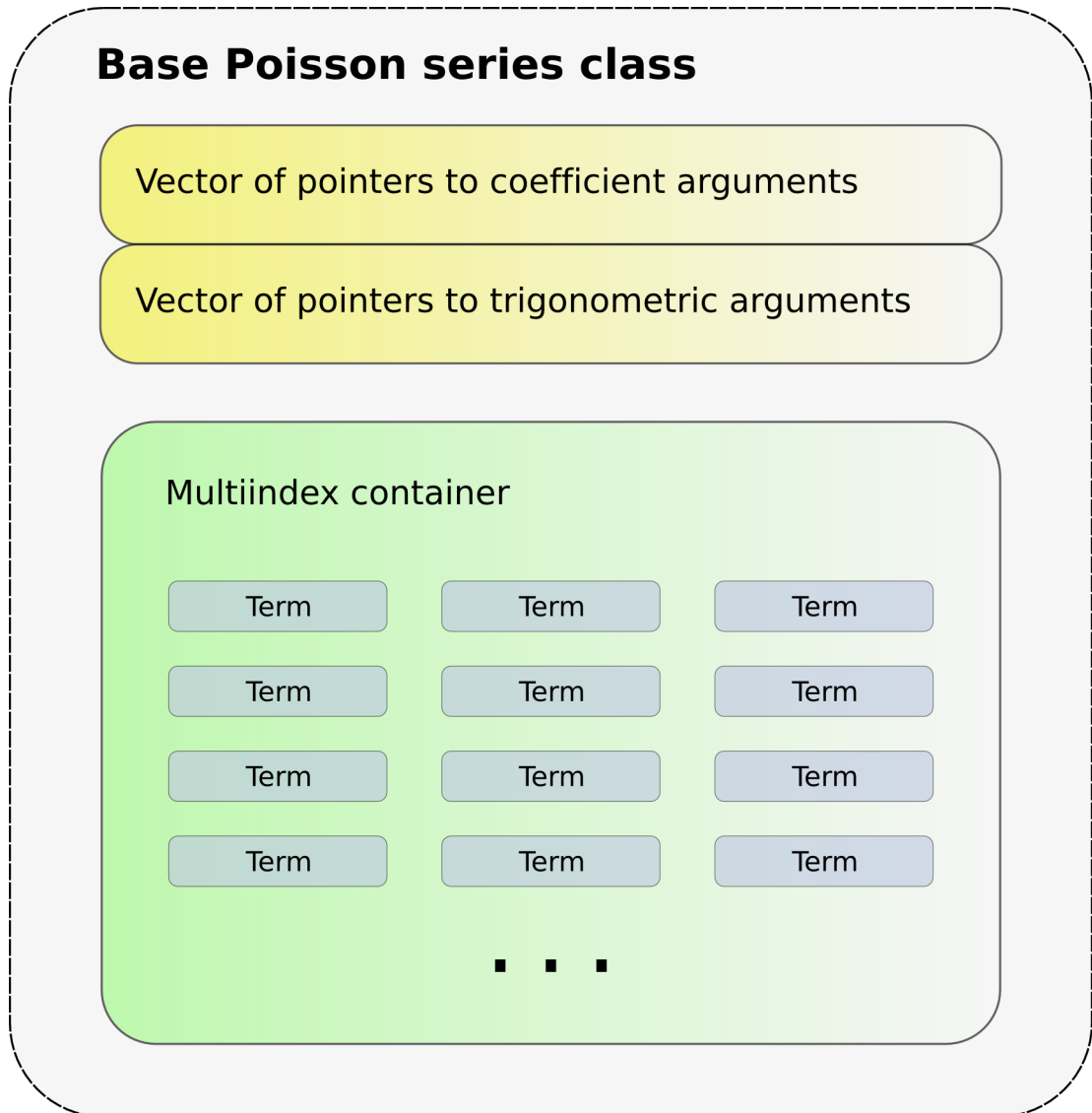


Figure 5.2: Base Poisson series class members.

interesting, for instance, to define an additional sorted index on the trigonometric parts with an ordering on the frequencies of the terms.

In Figure 5.2 the data members of the base Poisson series class are shown. It can be seen how the only members added with respect to a plain multiindex container are two vectors of pointers to the series arguments, whose sizes define the coefficient and trigonometric widths of the series. The representation and management of arguments are described in the next section.

5.3 Representation of arguments

Series arguments in Piranha are managed globally by a static class². Such manager class wraps a list of instantiations of the `psymbol` class, which serves as representation of the arguments of a Poisson series. The `psymbol` class is a simple structure which encapsulates two data members:

1. the name of the argument, represented as a standard C++ string,
2. the time dependence of the argument, represented as a C++ vector of double precision values.

The vector of double precision values can have an arbitrary size, and it contains the coefficients of the expansion in natural powers of time of the time-dependence of the argument. This means that the vector

$$(v_0, v_1, v_2, \dots, v_n)$$

maps to the time dependence

$$v_0 + v_1 t + v_2 t^2 + \dots + v_n t^n.$$

For trigonometric arguments this means that v_0 is the phase of the argument, while v_1 is its frequency.

When a new `psymbol` is created, the constructor registers the argument in the manager. If no other argument with the exact same name exists, a copy of the newly-created `psymbol` is appended to the list of known arguments, otherwise the time-dependence vector of the new `psymbol` replaces that of the corresponding argument in the manager's list.

As hinted earlier, base series classes do not store `psymbol` instances directly, instead they store vectors of pointers to existing arguments provided by the argument manager class. This way the weight of arguments in series classes is minimized and arguments consistency between series is ensured.

5.4 Toolboxes

In §5.1 we explained how the base series class, by implementing primitive operations on Poisson series, is not meant to be used directly, and that it is intended to be inherited by a derived class

²By *static class* we mean a class whose data members are static, which in turn means that its members are initialised when the program starts and that their existence does not depend upon any instantiation of the class.

5 Piranha: architecture and implementation details

which will implement higher level manipulations. While this approach ensures a clean architecture and a high degree of extendibility, it is also true that it places on the user's shoulders the burden of implementing the desired manipulation methods. It is also true that certain manipulation features are extremely convenient and can be generalised effectively: we are thinking, for instance, about operator overloading for common operations (addition, subtraction, etc.) or about those methods specific to the manipulation of series with complex coefficients (e.g., take the real/imaginary part of the series, calculate its complex conjugate).

One of Piranha's core principles is code reuse and genericity, which, in this context, translates in what we have called *toolboxes*. The idea is to use the ability of C++ to perform multiple inheritance, i.e. to inherit from more than one base class, and use generic methods-only classes which provide common higher-level functionality. Such toolboxes are inherited by the derived series class (which by definition already inherits from the base series class) to build a stack of customizable features. Toolboxes can use the methods from the base series class and reference its data members, and can also call methods from other toolboxes.

The introduction of toolboxes prompts for an update of the class diagram in Figure 5.1, which is displayed in Figure 5.3. The figure shows how the derived series class inherits from both the base series class and the toolboxes. It can be seen how toolboxes can be dependent upon each other, as indicated by the dashed arrows. The dependencies between toolboxes can be used to define toolbox concepts: the operators toolbox, for instance, needs the series addition and multiplication methods provided by the basic math toolbox; it could be then said that the operators toolbox implicitly defines a concept of which the basic math toolbox is a model. This also means that it is possible to write different implementations, for instance, of the basic math toolbox, without the need to change other toolboxes depending on the math toolbox itself. This is useful when we need to define different truncation methodologies for Fourier series coefficients (which are purely numerical) and full-fledged Poisson series coefficients (which are multivariate polynomials).

Piranha already includes many ready-to-use toolboxes, including³:

- an operator overloading toolbox, for comfortable use of ordinary mathematical operators (+, −, ·, etc.) in the manipulation of Poisson series,
- a basic math toolbox, implementing basic arithmetics,
- a trigonometric toolbox, implementing complex exponential, sine and cosine of Poisson series,
- a complex toolbox, providing methods specific to the manipulation of series with complex coefficients,
- a power toolbox, implementing the real exponentiation of series,
- a differential toolbox, implementing the partial derivation with respect to series arguments.

One of the design goals of Piranha is to encourage the development of extensions to the framework by its users, and the toolbox system aims to achieve precisely this goal.

³Some of the mentioned toolboxes are at the moment specific to Fourier series only.

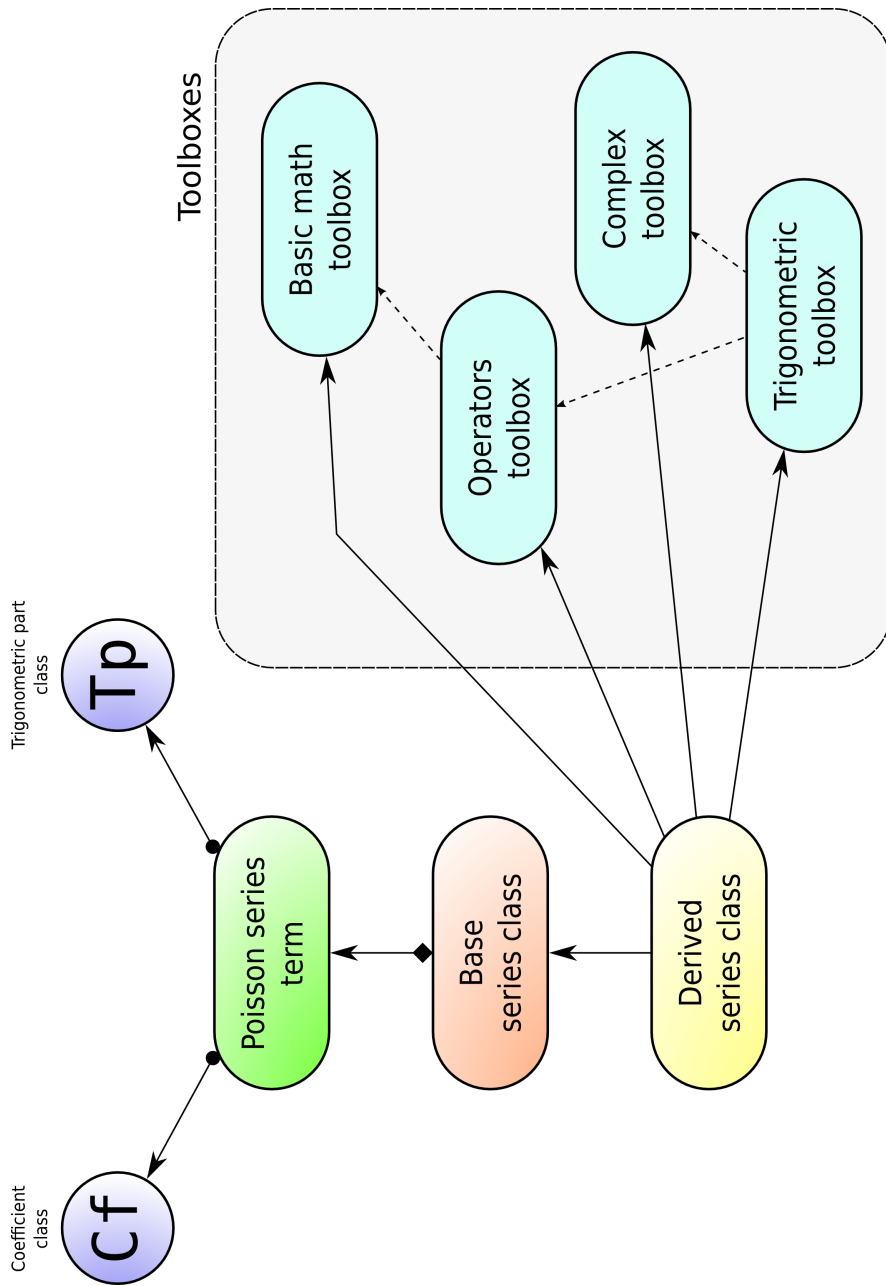


Figure 5.3: Relation between Piranha's base classes and toolboxes.

5.4.1 A note on the implementation of toolboxes

In order to reach maximum performance, the toolbox system is implemented in Piranha without any use of the runtime polymorphism capabilities of the C++ language. The type of polymorphism used in Piranha is strictly static (i.e., accomplished at compile time) and heavily relies on the design pattern known as *Curiously-Recurring Template Pattern* (or CRTP, see Coplien [1995]). This means, in practice, that from the point of view of the resulting binary this approach leads to the same result as if the methods defined in the toolboxes had been specified directly in the derived class by hand.

For the readers familiar with the Python programming language, the toolbox system can be seen as a set of C++ static, template-based mixins. Moreover, since toolboxes can be re-implemented and exchanged without changing other parts of the architecture, they can also be seen as a form of policy-based design (see Alexandrescu [2001] for a description of policy-based design patterns in C++).

5.5 Series I/O

In the typical use case of a Poisson series manipulator, series are stored as files on and loaded into memory from a mass storage device, typically a hard disk. The file format for Poisson series in Piranha is human readable and text-based. Basically, series are stored as a line-by-line sequence of terms in which coefficients and trigonometric parts are separated by a field-separator (specifically, a semicolon - “;”). A header, containing a description of the arguments of the series, is present at the beginning of the file. Lines beginning with the character “#” are considered comments and hence ignored.

The input routines try to be as fault-tolerant as possible: unknown entries in the header will be reported and ignored, trigonometric parts with more multipliers than those specified by the number of trigonometric arguments will be discarded, and so on. Here follows an example of Fourier series file:

```
1 # Main problem, latitude
2 [trig_arg]
3 name=theta_g
4 poly_eval=0.000000000000000e+00;2.301216752957500e+05
5 [trig_arg]
6 name=W_1
7 poly_eval=3.810344430588300e+00;8.399684731773899e+03
8 [trig_arg]
9 name=D
10 poly_eval=5.198466741027400e+00;7.771377146812100e+03
11 [trig_arg]
12 name=l'
13 poly_eval=6.240060126971500e+00;6.283019551679999e+02
14 [trig_arg]
15 name=l
```

5.6 Improving performance

```
16 poly_eval=2.355555898265800e+00;8.328691426955400e+03
17 [trig_arg]
18 name=F
19 poly_eval=1.627905233371500e+00;8.433466158130799e+03
20 [data]
21 8.950339292435590e-02;0;0;0;0;0;0;1;s
22 4.897463209452490e-03;0;0;0;0;0;1;1;s
23 4.846686200080110e-03;0;0;0;0;0;1;-1;s
24 3.023578483150850e-03;0;0;2;0;0;-1;s
25 9.671312989818789e-04;0;0;2;0;-1;1;s
26 8.075797467865170e-04;0;0;2;0;-1;-1;s
27 5.685003279693080e-04;0;0;2;0;0;1;s
28 3.001592522082110e-04;0;0;0;0;2;1;s
29 1.617213843065370e-04;0;0;2;0;1;-1;s
30 1.539760978998670e-04;0;0;0;0;2;-1;s
31 1.433978997103700e-04;0;0;2;-1;0;-1;s
32 7.546779444939431e-05;0;0;2;0;-2;-1;s
33 7.331182800950011e-05;0;0;2;0;1;1;s
34 -5.863676028915500e-05;0;0;2;1;0;-1;s
35 4.299584675330350e-05;0;0;2;-1;-1;1;s
36 3.858588454719500e-05;0;0;2;-1;0;1;s
37 3.604531541407670e-05;0;0;2;-1;-1;-1;s
38 -3.263634801535500e-05;0;0;0;1;-1;-1;s
39 3.189889792501930e-05;0;0;4;0;-1;-1;s
40 ...
```

This series represents the lunar colatitude in the main problem of the ELP2000 theory. Since this is a Fourier series, only trigonometric arguments will be present. The last element of each term line is a letter (“c” or “s”) that specifies whether the term is cosine or sine.

Poisson series can of course be saved back in this same file format, but they can also be saved (or displayed) in \TeX tabular format for quick embedding into other documents or to be processed and displayed as beautiful images in interactive sessions (currently, this is a work in progress).

5.6 Improving performance

In this section we are going to explain in greater detail how certain speed-critical parts of Piranha work, and the kind of optimizations employed to speed up such sections of code. More substantial performance optimizations (centered around the application of coded arithmetics techniques, see Chapter 6) were conceived after we wrote this part of the dissertation, and as such they are included in Chapter 9.

5.6.1 Use of temporary hash sets to speed up multiplications

In §2.3.1 we noted how multiplication is by far the most expensive mathematical operation on Poisson series. In the previous sections (see §5.2) we also mentioned how the core of the series classes is a multiindex container with two or more indices. The mandatory indices are a sorted index and a hashed index, whose performance profiles with respect to element insertion, lookup, modification and deletion are $O(\log n)$ and $O(1)$ respectively. This index layout creates a bottleneck during the sequence of operations performed during series multiplication, which we recall is the following:

1. scan the term sequence for both series and perform term-by-term multiplications;
2. insert in the output series the pair of terms resulting from each term multiplication; term insertion, as described in §2.3, relies on the ability to identify existing terms in the output series and either pack the coefficients in an appropriate way or append the new terms.

The performance critical part is here step two: while, by using the hashed index, we can quickly identify existing terms, when a term modification or insertion takes place the multiindex container must either reposition the reference to an existing term in the binary search tree or establish the placement of the newly-inserted term in the tree. Both such operations are characterized by logarithmic complexity.

To overcome this bottleneck, for series multiplications Piranha uses temporary hash tables, which, we recall, feature pure $O(1)$ performance in virtue of the fact that they do not maintain any ordering on the elements. This is not a problem, since during multiplication we are interested in identifying existing terms and pack them appropriately (the ordering is necessary in the factor series for the implementation of truncation methodologies); series ordering happens at the end of the multiplication when the terms are moved from the temporary hash table into the output series.

Such strategy makes sense just in series multiplication, which is a quadratic-complexity operation. Series addition, by having linear complexity and because it can be performed directly by modifying existing series, is not afflicted by the bottleneck described above.

In Piranha standard hashed containers, like `<ext/hash_set>` and `<tr1/unordered_set>`, are used as temporary hash tables during series multiplication. Performance can most likely be increased substantially by developing a custom hash table class specifically tailored on the specific task at hand.

5.6.2 Hash function

In the previous subsection we saw how in Piranha the efficiency of hash tables is critical for overall performance. Hash tables performance depends substantially on two factors:

1. the speed and behaviour of the hash function,
2. the strategy of collision resolution.

We want to focus here briefly on the properties of the hash function used in Piranha. Ideally, a hash function should produce values as randomized as possible. This means that a minimal change of the element being hashed should lead to a hash value different as much as possible from the original one. This ensures that elements are placed in the hashed container in a randomized fashion, hence minimizing the chance of collision and clustering behaviour (clustering happens when elements of the hashed container tend to group around specific positions). Moreover, a hash function will be used extremely frequently when working with a hashed container, and hence its speed becomes an important contributor to the overall performance of the program⁴.

The hash function used in Piranha is the one provided by the Boost libraries, which, in turn, was taken directly from Ramakrishna and Zobel [1997]. This hash function was conceived for the hashing of strings and, as shown in the cited paper, its performance in practice is very near to the theoretical best for a wide range of use cases. Besides, such hash function is very fast because it uses exclusively bit shift operations and integer additions, and it is so simple that it can be rewritten here:

$$s \oplus [n + 0x9e3779b9 + (s \ll 6) + (s \gg 2)], \quad (5.1)$$

where s is a seed value and n is the integer to be hashed. In this context \oplus is the bitwise XOR operator and \ll and \gg are the bit shift operators. $0x9e3779b9$ is a hexadecimal value (corresponding to decimal -1640531527). In Poisson series eq. (5.1) is used recursively on the members of the vectors of the trigonometric multipliers to build the hash value of such vector. The initial seed value is given by the type of the term (cosine or sine), which is a boolean flag (which, in C++, is a single-byte integer).

5.6.3 Packed operations on integers and SIMD instructions

Another common operation when dealing with hashed containers is the equality test for the elements in the container. As we saw in §4.5.2, indeed, the equivalence of the hash values of two objects does not imply the equivalence of the objects themselves, since there's the possibility of a hash collision. Hence, to establish if a term is already present in a series, a full-fledged comparison is often needed. Other frequent operations on elements of a series include:

- copy operations,
- assignment operations,
- tests on trigonometric multipliers vectors to establish whether they are null or not (we recall from §2.1.3 that in the case of a null sine trigonometric part the term is discarded upon insertion).

These operations (hashing, equality test, copy, assignment and null test on trigonometric parts) share a common feature: they can be performed in groups of integers instead of integer by integer.

Typical modern workstations, indeed, feature either 32-bit or 64-bit hardware integers. In the 32-bit case this means that integers in the range

$$[-2\,147\,483\,648, +2\,147\,483\,647]$$

⁴For a thorough discussion on the desired properties of hash function and hashed containers please refer to standard textbooks of computer science, such as Knuth [1998b].

5 Piranha: architecture and implementation details

can be represented. This is typically an overkill in the case of trigonometric multipliers of Poisson series, at least in the context of Celestial Mechanics. 16-bit or even 8-bit integers are all that's needed in most situations. A trick that can speed up many operations is then that of accessing arrays of short integers as if they were arrays of long integers instead, thus increasing the number of operations that can be performed during a single clock cycle. This technique is sometimes called *integer packing*.

As a practical example, let's take the case of null-testing an array of trigonometric multipliers:

$$(n_1, n_2, \dots, n_m) \stackrel{?}{=} \mathbf{0}.$$

Using the plain method, we need to perform at worst m integer comparisons for the null test itself, plus m integer additions and further m comparisons in the cycle that iterates over the array. We can employ integer packing in this context, since if $n_1 = n_2 = 0$ then also the union of these short integers results in a null long integer, whereas if n_1 or n_2 is not zero the resulting long integer won't be zero either. If the size of the short integer is half the size of the long integer (e.g., packed 16-bit against unpacked 32-bit), then for the same null test half comparisons and half integer additions will be needed with respect to the plain method. In the extreme case of 8-bit integers and a 64-bit workstation, an array of eight elements can be null-tested in a single pass.

Such technique of integer packing is made portable by the Boost integer library, which allows to define integers of arbitrary sizes independently of the hardware platform. Piranha, hence, will use integer packing in an optimal way on both 32-bit and 64-bit architectures without the need to change any line of code. According to our testing, the use of operations on packed integers in Piranha leads to performance improvements estimated in the 20% - 40% range during common operations on Poisson series.

The concept of integer packing can be extended also to those operations which are not naturally suited for it. Most notably in this context we are interested in speeding up additions and subtractions of trigonometric multiplier vectors, which are performed during series multiplications. Special *vectorizing* (or *SIMD* - Single Instruction, Multiple Data) instructions exist for many architectures:

- MMX/SSE for Intel processors,
- AltiVec for PowerPC processors,
- 3DNow! for AMD processors,

and so on. Typically these instructions allow to perform mathematical and logical operations on vectors of integers (and also floats) at the cost of a single operation, with theoretically high performance gains. In practice, the vectorizability of a problem depends on certain constraints such as proper memory alignment, and it is also dependent upon the minimization of the cost of packing/unpacking of data to/from the special registers used by SIMD extension.

In Piranha support for Intel's SSE2 (Streaming SIMD Extensions 2) can be enabled at compile time. SSE2 instructions are used during the multiplication of trigonometric parts, and from our measurements they lead to a performance gain of about 10%. More substantial gains are expected when SIMD instructions will be applied in the context of coded arithmetics (see §9.1).

5.6.4 Memory management

Memory management is typically a critical area for software performance. This is particularly true for Piranha, where multiindex containers are widely employed. Multiindex containers, indeed, alongside with their effective payload (i.e., the terms of a series) must also maintain two or more indices by using iterators (see §4.3.5) and tree-like structures. Iterators are typically small-sized (few bytes), and hence it becomes important to be able to quickly allocate small sized objects dynamically (i.e., on the heap).

Luckily, multiindex containers (as well as standard C++ containers like vectors, hash tables, trees) allow to specify as optional template parameter a memory allocator different from the predefined one. GCC's C++ standard library (libstdc++) provides allocators tuned for specific workloads; in particular in Piranha we use extensively the pool allocator provided by the `<ext/pool_allocator.h>` header (see libstdc++ allocators). A pool allocator allocates large chunks of memory which are then subdivided and provided upon request, whereas the standard memory allocator typically performs one allocation per request. Pool allocators are hence suited for those situations in which it is needed to allocate many objects of small and fixed size, just like in multiindex containers.

Through the use of this pool allocator we were able to cut down the number of memory allocations during typical manipulations by at least an order of magnitude. Moreover, since libstdc++ is Free Software, we were able to incorporate the source code of the pool allocator (which is quite small) into Piranha, and modify it so that it can allocate memory properly aligned for usage with SSE2 instructions.

Thanks to the standardized interface of C++ memory allocators it is possible to write a customized memory allocator that can be plugged inside Piranha's architecture without changing any line of code. A possible future optimization in Piranha is then the implementation of a pool allocator heavily oriented towards use in Poisson series and polynomials.

5.6.5 Improving evaluation speed

An operation which is often performed in certain manipulations, especially on Fourier series, is series evaluation. Series evaluation is the process of associating a numerical value to the series by substitution of literal arguments with numerical values. Evaluation is used for instance to assess the magnitude of terms during Taylor-like expansions, or to calculate the position of a body using a theory of motion expressed as Fourier series.

The straightforward, brute-force, algorithm for series evaluation in Piranha is the direct substitution of time-evaluations of the `psymbol` objects inside the coefficients and trigonometric parts of the series, the computations of cosines and sines, and the summation of the evaluations of all the terms. This method is not optimal, performance-wise, because it forces many costly calls to cosine and sine functions that can be avoided.

The strategy adopted in Piranha is to transform the trigonometric parts, during evaluation,

5 Piranha: architecture and implementation details

into products of complex exponentials, i.e.,

$$\begin{cases} \cos \\ \sin \end{cases} (n_1 a_1, n_2 a_2, \dots, n_m a_m) = \begin{cases} \Re \\ \Im \end{cases} \left[e^{i(n_1 a_1, n_2 a_2, \dots, n_m a_m)} \right] \quad (5.2)$$

$$= \begin{cases} \Re \\ \Im \end{cases} \left[\prod_{j=1}^m e^{i n_j a_j} \right], \quad (5.3)$$

so that it is possible to cache all the individual complex exponentiations inside a vector-like structure and re-use them throughout the evaluation of all terms. This way the vast majority of calls to cosine and sine functions are replaced with faster complex multiplications.

This approach leads to substantial improvements performance-wise, as shown in Figure 5.4. The graph depicts three different evaluation tasks performed with three different algorithms:

1. the “dumb” brute-force algorithm (red),
2. a smarter brute-force algorithm that avoids calculating the contribution of null trigonometric multipliers (green),
3. the complex exponential caching algorithm (blue).

The first cluster of bars refers to the execution time for 200000 evaluations of the ELP3 lunar series, which counts 702 terms. The second cluster of bars depicts the execution time of 1000 evaluations of a series (here called TASSR6) derived from the TASS theory for Saturn’s satellites by Alain Vienne, representing Titan’s distance from Saturn and counting around 16000 terms (more about that in Chapter 8). BIGSERIES is a precise squaring of the TASSR6 series (numerical precision of 1 part every billion), it counts around 300000 terms and it is evaluated 100 times.

The caching algorithm leads to a speedup in the range of x1.5 - x3 with respect to the brute-force algorithms, and hence it is a marked improvement. A yet more efficient approach is probably the one described in Babaev et al. [1980]: this algorithm groups in common radix fashion the vectors of trigonometric multipliers (i.e., it places them in a trie-like structure – see Knuth [1998b] for a description of the trie data structure), effectively caching complex exponentials of entire linear combinations of trigonometric multipliers instead of complex exponentials of single multipliers. This leads to a further reduction in the number of operations needed during evaluation with respect to our caching algorithm. Such evaluation algorithm will be implemented in the future also in Piranha.

5.6.6 Parallelization

Parallelization is the procedure of subdividing computational tasks between logical processing units, so that such processing units can work simultaneously. Such processing units can be multiple cores inside a single physical processor, multiple processors in a single workstation and multiple workstations connected in a network. Multi-core and multi-processor parallelization are usually the simplest for to implement, since they can use shared memory, and require less modifications to the source code with respect to network parallelization.

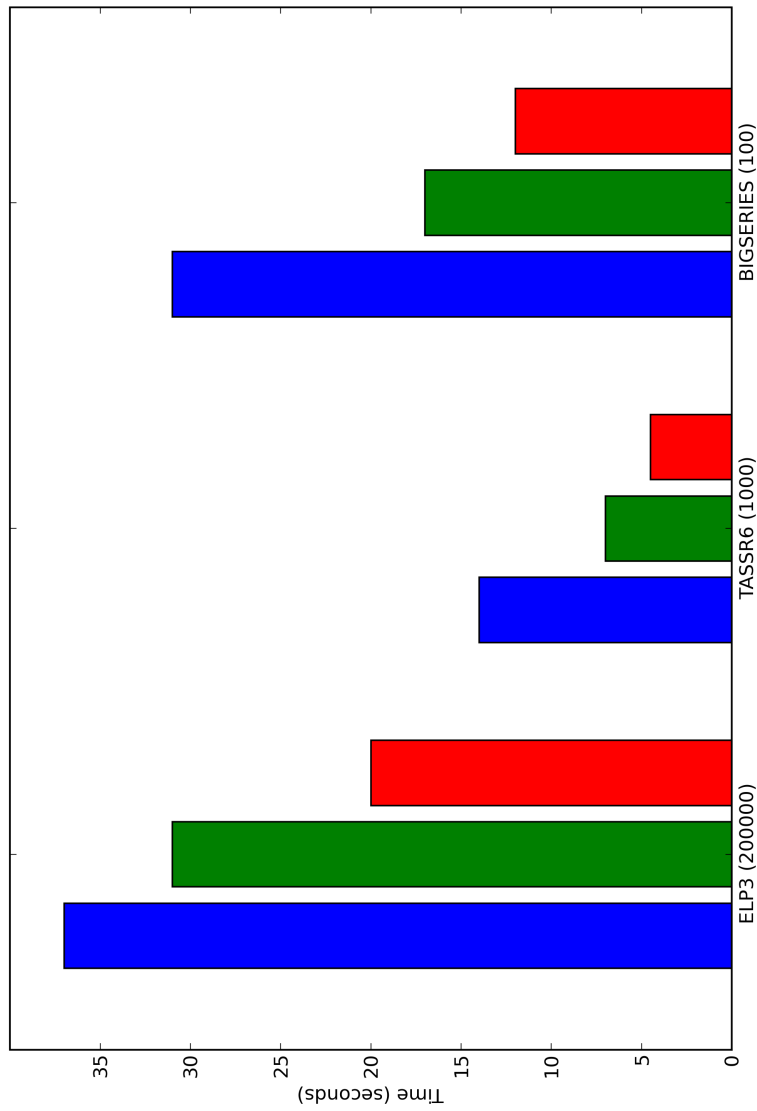


Figure 5.4: Benchmark for three evaluation algorithms: dumb brute force (red), smarter brute force (green) and complex exponential caching algorithm (blue). The series being evaluated are taken from the TASS and ELP2000 theories.

5 Piranha: architecture and implementation details

In Piranha we have started to implement multi-core/multi-processor parallelism using Intel's threading building blocks (TBB) library (see TBB). TBB, which was recently made available under an open-source license, allows programming in modern C++ style (using idioms like template classes, functors, etc.), and hides much of the complexity and minute details of multi-threaded programming from the user. This is a marked advantage, in a C++ project like Piranha, with respect to other solutions like low-level thread programming or OpenMP (see OpenMP), which are based on C-style programming. With TBB it is possible to add parallelism with a very low impact on the code base.

At the present time parallelism in Piranha is available only in the evaluation methods, which are algorithmically quite simple and do not require locking techniques for access to shared resources from different threads. A glimpse on the kind of performance improvements that can be expected through parallelism is shown in Figure 5.5. The graph refers to the time-evaluation of a series performed over a certain time-span with around 300000 samples over such time span. This test was performed in serial (blue columns) and parallel (green columns) mode on a dual-core 64-bit laptop and on a 4-core (2 CPU x 2 cores) Xeon server, and compared to the theoretical maximum performance achievable on such hardware (red columns), which equals to the serial time divided by the number of logical processing units. As shown in the graph, on the dual-core laptop the actual parallel performance is very near to the maximum theoretical performance (i.e., running time is divided almost by half). On the 4-way Xeon parallel performance is not so close to the best theoretical performance (it is ~30% away), but a substantial speed boost is gained nevertheless (we don't know yet if this is due to some overhead due to TBB or to cache effects, most likely a combination of the two - it can certainly be improved).

A current work in progress is the extension of parallelization to other algorithms in Piranha, most notably to the cumbersome multiplication methods. Parallelization is expected to bring substantial performance improvements, especially when used in conjunction with coded arithmetics (see Chapters 6 and 9).

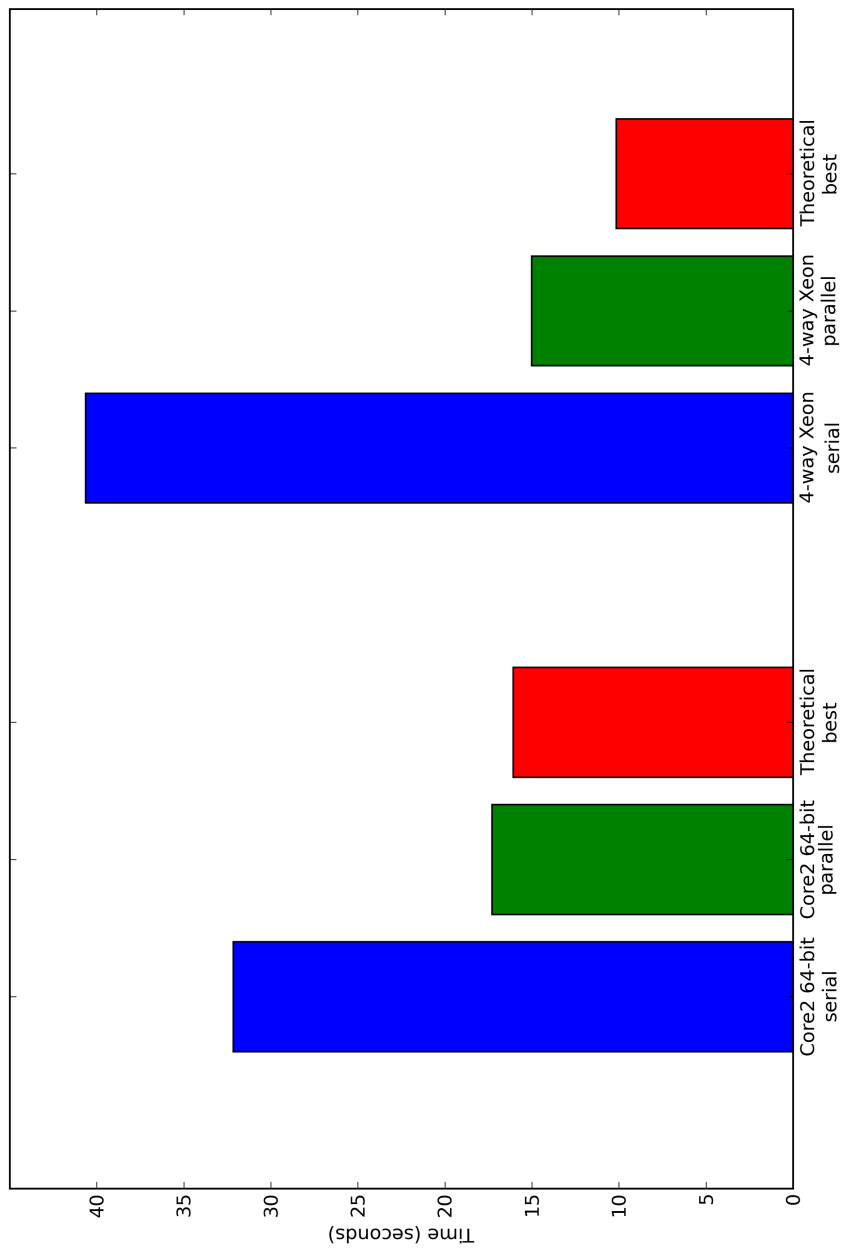


Figure 5.5: Benchmark of parallelized evaluation. Timings for 300000 evaluations of a Fourier series on two different multi-core system in serial and parallel mode are displayed, and compared to the theoretical minimum running time in parallel mode.

5 Piranha: architecture and implementation details

6 ON THE MANIPULATION OF SPARSE MULTIVARIATE POLYNOMIALS

IN the previous chapters we have focused mainly on the manipulation of Fourier series and of the trigonometric parts of Poisson series. The reasons for this choice are essentially two: in the first place, Piranha was conceived initially as a Fourier series manipulator, and secondly sparse multivariate polynomials (the polynomials we are mostly concerned with – see §6.1.2) can efficiently be handled using idioms similar to those employed in the manipulation of the trigonometric parts of Poisson series.

Many polynomial manipulators are available today, and projects like Pari and Singular (see PARI/GP and Greuel et al. [2007]) provide featureful environments for mathematicians; such manipulators are however often geared towards the resolution of theoretical problems, and as such they often compromise on pure performance to allow more flexibility. As an illustrative example, we cite the fact that manipulators geared towards number theory often employ arbitrarily-sized integers for the representation of polynomial exponents. In Celestial Mechanics hardware integers are more than enough for the task.

Additionally, many of the advanced capabilities offered by specialised polynomial manipulators are not used in Celestial Mechanics: the main goal is to perform simple operations on very large polynomials as fast as possible.

6.1 Types of polynomials

In the context of computer algebra systems polynomials are classically subdivided into two groups: *dense* polynomials and *sparse* polynomials.

Given a degree n , a polynomial is defined dense when it includes all (or almost all) the monomials from degree 0 up to degree n . Conversely, a polynomial of degree n is defined sparse when most of the monomials are null.

From a mathematical point of view, hence, there is no difference between dense and sparse polynomials. Fundamental differences arise however in the representation and manipulation of dense and sparse polynomials in computer algebra systems.

6.1.1 Dense polynomials

Dense polynomials can intuitively be represented efficiently by simple arrays. To represent a univariate dense polynomial of degree n , for instance, it is enough an array of $n + 1$ elements which stores the coefficients of the monomials; the position of each coefficient in the array will also be the degree of the corresponding monomial:

C_0	C_1	C_2	\dots	C_n
-------	-------	-------	---------	-------

For the representation of multivariate dense polynomials, table-like or recursive representations are often used. Some existing computer algebra systems pre-compute tables of integers (either at startup or on-demand) in which each row corresponds to a multiindex of exponents. The position of the coefficient in an array gives implicitly the row corresponding to the monomial's multiindex. This way the amount of data being manipulated is reduced, the exponents multiindices can be shared between polynomials and the cache memory utilisation is optimized. This approach is notably used in one of the possible polynomial representations of the algebraic manipulator TRIP (see Gastineau and Laskar [2005]).

The manipulation of dense polynomials has been studied extensively in literature, and as a result a considerable number of techniques is available when dealing with such objects. Both polynomial long division and evaluation, for instance, can be performed using the well-known Horner scheme (see Horner [1815]), which, as shown in Ostrowski [1954], is an optimal algorithm for the evaluation of univariate polynomials.

Most notably, fast algorithms for dense polynomial multiplication have been developed. Polynomial multiplication, like Poisson series multiplication, has $O(n^2)$ complexity when performed straightforwardly (i.e., monomial-by-monomial multiplication). Karatsuba and Ofman [1963] were the first to introduce an algorithmically faster multiplication methodology, which has a complexity of $O(n^{\log_2 3}) \sim O(n^{1.585})$. A yet better approach, at least from a theoretical point of view, is to use FFT-based multiplication methods, which have logarithmic complexity (see, for instance, Brigham [1988] and Emiris and Pan [1999]).

Despite their theoretical advantages, the use of these multiplication methodologies in Celestial Mechanics is however not very effective: Karatsuba and FFT methods are advantageous from polynomial orders typically higher than those seen in Celestial Mechanics, since, for lower polynomial orders, the implementation cost of these algorithms outweighs their lower big O complexity (see Moenck [1976] for a thorough comparison of polynomial multiplication algorithms).

Horner's scheme, as well as Karatsuba and FFT multiplication, are suitable for use in univariate polynomials, but they can be adapted to work also on multivariate polynomials through the recursive representation of multivariate polynomials as univariate polynomials with multivariate polynomials as coefficients.

6.1.2 Sparse polynomials

Although they can be applied also to sparse polynomials, the techniques mentioned in the previous subsection are often ineffective for such objects. The fast multiplication algorithms, for instance, work best on the assumption that the polynomials being handled are dense, on the basis that the position of each monomial in the resulting polynomial is implicitly given in the dense structure and hence all CPU time is spent in coefficient multiplication. As shown in Fateman [2003], Karatsuba multiplications (and FFT likewise) on sparse polynomials typically imply *more* coefficient multiplications than the straightforward method, since it will be needed to deal also with the monomials missing in a sparse polynomial.

The focus on the manipulation of sparse polynomials shifts hence to the data structures used to

represent them. In the following section we will discuss in more depth the design choices made in Piranha regarding the manipulation of polynomials.

6.2 Polynomials in Piranha

Sparse multivariate polynomials are not much different from Poisson series when it comes to algebraic manipulation. The issues arising for the efficient manipulation of polynomials are the same analysed in §2.3.1, with the difference that instead of trigonometric terms we are dealing with monomials, and hence a monomial-by-monomial multiplication generates a single monomial (whereas trigonometric part multiplication leads to two terms being generated). The analogy between polynomials and Poisson series is so appropriate that in some existing Poisson series manipulators the same data structures are used for polynomials and trigonometric parts, and in certain cases Poisson series are represented as polynomials of complex exponentials (see, for instance, the discussion in Cherniak [1970]).

In Piranha we chose to focus on sparse polynomials for the sake of generality. By this we mean that we chose to provide two already implemented sparse polynomial classes: it is of course possible to leverage the infrastructure provided by Piranha's architecture to write a dense polynomial class and use it as coefficient in Poisson series (see the discussion in Chapter 5).

6.2.1 A general-purpose polynomial class

The first sparse polynomial class we implemented mirrors almost exactly the design of Poisson series classes described in Chapter 4. In brief:

- monomials are classes containing a coefficient and an array of integers (i.e., the exponents),
- monomials are stored in a `boost::multi_index_container` class with a hashed-sorted index,
- the hashed index¹ is based on the values of the exponents (see §5.6.2),
- the sorted index is based on the monomials' degree,
- the same optimization techniques described in §5.6 for series multiplication are applied for polynomial multiplication.

This polynomial class is generic in the sense that it does not impose hard limits, apart from those fixed by the ranges of the types used in the representation: the number of monomials that can be contained in a polynomial is arbitrary, and so is the value of the exponents.

¹We note here that hashing techniques, while common in more general polynomial manipulators (see for instance Hall Jr. [1971] and Fateman [2003]), seem strangely uncommon in specific algebraic manipulators for Celestial Mechanics.

6.3 A faster polynomial class: coded monomial arithmetics

We have briefly mentioned in §6.1.1 that a common technique in the manipulation of polynomials is to build tables of exponents and arrays of coefficients, and relate them through row numbers and positions respectively. Such tables of exponents are usually built in lexicographic fashion and grouping together rows with the same degree (a representation that is sometimes called *total degree lexicographic ordering*, which, according to Bachmann and Schönemann [1998], was first implemented in the Macaulay algebraic manipulator – see Grayson and Stillman [2007]).

With three variables and up to polynomial degree three, for instance, we may write the following table of exponents:

x	y	z
0	0	0
0	0	1
0	1	0
1	0	0
0	0	2
0	1	1
0	2	0
1	0	1
1	1	0
2	0	0
0	0	3
...
3	0	0

Please note that the monomials are sorted in ascending order with respect to the degree. The monomial

$$5xz, \tag{6.1}$$

for instance, is densely represented by the slot number 7 in the array of coefficients, which will contain the value 5. A sparse representation may instead store the coefficient in a list-like structure alongside with a pointer to the appropriate multiindex row.

Building such tables takes time and, most importantly, they utilise a lot of memory storage. This is not a problem for dense polynomials – since by definition they will comprise most or all the possible monomials – but it becomes an issue for sparse polynomials.

An alternative is not to store explicitly all the monomials and use instead some kind of codification. For instance:

6.3 A faster polynomial class: coded monomial arithmetics

x	y	z	Code
0	0	0	0
0	0	1	1
0	1	0	2
1	0	0	3
0	0	2	4
0	1	1	5
0	2	0	6
1	0	1	7
1	1	0	8
2	0	0	9
0	0	3	10
...

Since there is a bijective relation between each exponent multiindex and the corresponding coded value, it will be enough to store the code alongside with the coefficient to represent a monomial. The downside is that when manipulating such objects it will be typically needed to decode the coded value, perform the operation, and re-encode the new monomial. Typically the operations of (de)codification will involve combinatorics and will occupy a considerable amount of CPU time (as shown, for instance, in Jorba [1998]).

Here we describe a polynomial class that takes advantage of the coded representation without the need for (de)codification during basic algebraic operations (i.e., addition, subtraction and multiplication).

Note *We initially conceived this schema after we attended Prof. Gastineau's lectures during a school on specific algebraic manipulators in Barcelona (SAM07). Despite our attempts at locating precedent descriptions of this technique in the literature, it was not until the days immediately preceding the printing of this dissertation that we discovered that this procedure is a variant of the Kronecker's algorithm, which is described, for instance, in Fateman [2005] and Moenck [1976]². We apologize, hence, for calling here "coded arithmetics" a technique already known with another name: due to the tight time constraints we could not alter too much the dissertation without the risk of rendering it confusing.*

The basic idea is to use a coded representation with different ordering with respect to the one shown above. A 3-variate polynomial of degree three, for instance, is represented as follows:

²We are, however, reasonably sure that the application of Kronecker's algorithm to Poisson series (which we outline in §9.1) has not been described in any publication connected to algebraic manipulators specialised for Celestial Mechanics.

6 On the manipulation of sparse multivariate polynomials

z	y	x	Code
0	0	0	0
0	0	1	1
0	0	2	2
0	0	3	3
0	1	0	4
0	1	1	5
0	1	2	6
0	1	3	7
0	2	0	8
0	2	1	9
0	2	2	10
0	2	3	11
...
3	3	3	63

The order is still lexicographic, but there is no grouping of monomials with equal degree and no ordering with respect to monomial degree. Please also note that the order of the columns is inverted. We can call this representation *m-variate n-lexicographic*, where, in this case, $n = m = 3$. The following features of such a representation are evident:

- it can represent all 3-variate polynomials of degree three,
- it can represent a subset of polynomials of degree greater than three.

In other words, this representation is less compact than the total degree lexicographic ordering, which does not include monomials of degree greater than three. This representation, though, has an interesting feature. If we consider, for instance, the monomials M_4 and M_5 , corresponding to codes 4 and 5, we notice that the code corresponding to the monomial resulting from the multiplication of M_4 and M_5 corresponds to code $9 = 4 + 5$. In other words:

$$M_4 \cdot M_5 \rightarrow \underbrace{(0, 1, 0)}_{\text{code 4}} + \underbrace{(0, 1, 1)}_{\text{code 5}} = \underbrace{(0, 2, 1)}_{\text{code 9}}. \quad (6.2)$$

This means that we can perform monomial multiplication on the coded values instead of the multi-indices exponents. This way the complexity of exponent manipulation during the multiplication of two m -variate monomials is reduced from $O(m)$ to $O(1)$, and, perhaps more importantly, we can avoid the (de)codification step in polynomial multiplication. This technique works as long as the monomial resulting from the multiplication can still be represented in the above table, or, in other words, all the exponents of the resulting monomial are less than or equal to the degree of the representation. If we try to multiply the monomials coded by the numbers 6 and 7, for instance, the resulting monomial will feature an exponent greater than three and hence it will not be able to be represented in the lexicographic table eq. (6.2) will not hold.

To generalize the above results, we give the following definitions.

6.3 A faster polynomial class: coded monomial arithmetics

Definition Given an m -variate n -lexicographic representation of a set of monomial multiindices

$$\mathbf{j}_k = (j_{0,k}, j_{1,k}, j_{2,k}, \dots, j_{m-1,k}), \quad (6.3)$$

we call *coding vector* the integer vector

$$\mathbf{c} = (1, n+1, (n+1)^2, \dots, (n+1)^{m-1}), \quad (6.4)$$

and *lexicographic code* the dot product

$$\mathbf{h}_k = \mathbf{c} \cdot \mathbf{j}_k. \quad (6.5)$$

Given three multiindices \mathbf{j}_1 , \mathbf{j}_2 and \mathbf{j}_3 that can be represented in a m -variate n -lexicographic representation and for which the relation

$$\mathbf{j}_1 + \mathbf{j}_2 = \mathbf{j}_3 \quad (6.6)$$

holds, it is then evident that the same relation holds when both sides of the equation are dot-multiplied by the constant coding vector \mathbf{c} ,

$$\mathbf{c} \cdot (\mathbf{j}_1 + \mathbf{j}_2) = \mathbf{c} \cdot \mathbf{j}_3, \quad (6.7)$$

hence leading to the equality of the corresponding lexicographic codes:

$$\mathbf{h}_1 + \mathbf{h}_2 = \mathbf{h}_3. \quad (6.8)$$

It is fairly easy to show that the codes \mathbf{h}_1 , \mathbf{h}_2 and \mathbf{h}_3 are unique in the representation, given an appropriate coding vector.

Applying the coding vector to a multiindex can be seen as considering the elements of the multiindex as the digits of a number in base $n+1$, and transforming it into its decimal counterpart. In the 3-variate example above, for instance, the last element of the table, 333, is decimal 63 in base 4. In other words, we map the vectorial space of the \mathbf{j}_k vectors to a subset of the non-negative integers. Such mapping is a homomorphism which preserves the operation of addition.

Decoding back a lexicographic code into the corresponding multiindex is a matter of applying multiple times the modulo operation (which we note with the C-style “%” symbol):

$$\mathbf{j}_k = \left(\frac{\mathbf{h}_k \% (n+1)}{(n+1)^0}, \frac{\mathbf{h}_k \% (n+1)^2}{(n+1)^1}, \dots, \frac{\mathbf{h}_k \% (n+1)^m}{(n+1)^{m-1}} \right). \quad (6.9)$$

We note that the last element of the vector can be simplified into

$$\frac{\mathbf{h}_k}{(n+1)^{m-1}}, \quad (6.10)$$

since by definition

$$\mathbf{h}_k < (n+1)^m \quad \forall k. \quad (6.11)$$

The advantages of this monomial representation are the following:

6 On the manipulation of sparse multivariate polynomials

- the exponent part of multivariate monomial multiplication is performed in $O(1)$ (constant time),
- it is not necessary any more neither to store explicitly the multiindices nor to employ coding/decoding functions during multiplication, and hence the memory footprint and CPU utilization are drastically reduced.

The main disadvantage of this representation is that there is a limit in the number of non-negative (or *unsigned*) integers that can be represented on a real computer. On a 32-bits machine the unsigned integer interval

$$[0, 2^{32} - 1 = 4\,294\,967\,295]$$

can be represented in hardware. The number of elements of a n -lexicographic representation for m -variate monomials is easily calculated as

$$(n + 1)^m, \tag{6.12}$$

which, for 32-bits architectures, means that n and m are bound by the following relation:

$$n \leq 2^{\frac{32}{m}} - 1. \tag{6.13}$$

To put this in perspective with an example, in the case of studies about dynamical systems, which often employ 6-variate polynomials, this means that a maximum degree of 39 can be reached. On 64-bits machines, the limit becomes degree 1624.

It is of course possible to use double-word unsigned integers to augment the range of the representation. Doing so however has a noticeable negative impact on performance. An interesting possibility would be to use a big integer library, like GMP, to represent the lexicographic codes in case of sparse polynomials with a very high number of variables. This approach would likely become advantageous as soon as the cost of m integer additions became higher than a big integer addition.

6.3.1 Implementation of a sparse polynomial class with coded arithmetics

The coding technique described in the previous section lends itself nicely for the representation in hash tables (see §4.5.2 for an overview). Lexicographic codes, indeed, are themselves *perfect* hash functions, in the sense that the mapping between a monomial and its lexicographic code is univocal.

This leads to two additional advantages of the coded polynomial class over the general polynomial class:

- the operations performed for lookup and insertion in a hash table are reduced in complexity from $O(m)$ to $O(1)$,
- an optimized perfect hash table can be implemented.

The first point means that if we store monomials as coefficient-lexicographic code pairs in a hash table, then the computation of the hash values is performed simply by reading the code value, and the comparison between monomials is done simply by comparing the lexicographic codes. By contrast, in the more general polynomial class described above the same operations involve manipulations and comparisons on each of the m exponents. This complexity reduction is very beneficial for performance, since typically during polynomial multiplications a lot of time is spent looking for elements in the hash table and updating them. Additionally, the reduced memory footprint of the coded representation allows a better usage of cache memory.

Lexicographic codes can also be used as indices in an array (i.e., as in a *lookup table*). This is clearly less compact than a hash table, since it requires the allocation of memory for all the monomials admitted in the representation, but leads to higher performance since the overhead related to the management of the hash table is avoided.

6.4 A mixed approach?

A possibility to retain the speed of the coded representation and the flexibility of the general polynomial class is to implement the coded representation only when performing polynomial multiplications. The overhead of this operation would be minimum, since the complexity signature would be linear $O(n)$ (while, we recall, the complexity of polynomial multiplication performed with the straightforward method is $O(n^2)$). Once the multiplication is performed, the result would be decoded back in the general polynomial class.

Additionally, the coded representation would be used only when possible (i.e., when the polynomial degrees are small enough), otherwise the standard multiplication would be employed instead.

With such an approach, finally, it would be possible to tailor the coding of the polynomials on a case-by-case basis, which in practice would mean to enlarge the applicability of coded arithmetics. The idea is to analyze the polynomials involved in the multiplication, establish the exponents' upper limits and use a customized coding vector which respects the exponents limits (instead of the coding vector proposed above, which is a vectorial function of the sole polynomial degree).

These ideas are currently being tested and implemented in Piranha, and constitute a work in progress. A procedure for the generalization of coded arithmetics on lexicographic representations is sketched in §9.1, where it is shown that such a technique can lead to substantial performance benefits in the manipulation of Poisson series (see §9.2.1), and a flexible multiplication algorithm for Poisson series based on coded arithmetics is presented.

6 On the manipulation of sparse multivariate polynomials

7 PYRANHA, THE PYTHON BINDINGS FOR PIRANHA

Make simple things easy, and difficult things possible.

– *Perl motto*

PIRANHA is designed as a C++ library, which means that to use it a C++ `main()` routine has to be written. This routine, which will contain instantiations of Poisson series and all kind of manipulations, will have to be compiled and run, and may look like this:

```
1 #include "piranha.h"
2
3 using namespace piranha;
4 using namespace std;
5
6 int main()
7 {
8     np elp3("elp3.csv");
9     np elp3_sqinv=elp3.pow(-.5);
10    // Print to screen the result.
11    elp3_sqinv.put();
12    // Save the series to a file.
13    elp3_sqinv.save_to("elp3_sqinv.csv");
14    np elp1("elp1.csv");
15    np elp1_cos=elp1.cosine();
16    elp1_cos.save_to("elp1_cos.csv");
17    // Normal exit.
18    cout << "Everything ok, returning." << endl;
19    return 0;
20 }
```

If we want to change the series of operation performed, we will have to change the source code, recompile and re-run.

This workflow is evidently not very smooth and flexible:

- it requires the user of the software to know at least the basics of the C++ language,
- it involves frequent recompilations, which in turn require the availability of a development environment and, especially in case of template programming, a lot of time and memory;

7 Pyranha, the Python bindings for Piranha

- the use of the manipulator is not interactive: if we need to change the manipulations to be performed on series we cannot retain already computed results, and we have to start from scratch. This can become quite a burden, especially when dealing with long, time-consuming computations.

In this short chapter we will briefly present the solution adopted in Piranha to facilitate the access to its manipulation capabilities.

7.1 Easing the utilisation of specific manipulators

For the reasons explained above, practically every Poisson series manipulator employs some kind of high-level interface to its capabilities. In the early days of computer algebra higher level languages or language extensions which exposed the grammar of Poisson series manipulation were built on top of FORTRAN or even in assembly. Example of these approaches can be seen, for instance, in Brown [1966] and Bourne and Horton [1971]. More recently, many specialized manipulator typically offer some kind of interactive interface (often a command prompt) and their own scripting language. This approach is also adopted in commercial general-purpose manipulators, like Mathematica and Maple.

7.1.1 Issues with existing approaches

We believe that the approach adopted by existing manipulators (i.e., command prompt and proprietary scripting language), while certainly fulfilling the goal of an easier and more flexible access to the manipulation capabilities, has some flaws and can be improved. In our opinion the main flaws are the following:

- custom languages are not “real” languages: by this we mean that, first of all, they are not formalised. This becomes an issue especially in commercial, general-purpose manipulators, where the syntax of the language often fluctuates between releases and there is not a solid commitment to backwards compatibility. As a software package written in these custom languages grows, the chance of introduction of incompatible changes in the language grows as well, with the bleak perspective of ending up locked to a particular revision of the manipulator. This is especially true in commercial manipulation packages, where a vendor-controlled language may be used as a mean to “encourage” upgrades to the latest version. Secondly, such languages are often tailored for the specific task of the manipulator. As the manipulator grows and acquires new capabilities, a rudimentary language may become a burden: writing a good programming language is an extremely difficult task;
- it won't be possible to share the syntax of the language across manipulators, since each software package will implement its own construct, keywords, grammar, etc. This limits the possibility of interoperability between manipulators.

The solution we adopted, consistently with the commitment to openness and leverage of existing solutions on which Piranha is based, is to expose Piranha's capabilities in the Python program-

ming language¹.

7.1.2 The Python programming language

Python (see Python) is a widely used high-level interpreted language, with a clean, powerful syntax and a low learning curve; it is designed around a philosophy which emphasizes readability and the importance of programmer effort over computer effort. Python core syntax and semantics are minimalist, while the standard library is large and comprehensive. Python is a mature project and it is used pervasively in various production and mission-critical environments.

It is possible to *wrap* (or *expose*) C and C++'s routines and classes to make them accessible from Python. This is not a translation from one language to another: the routines are still compiled into binary code, but they can be invoked from Python. Their speed, hence, will be unaffected, but their functionalities will be accessed by a higher-level interpreted language. This approach is very flexible, since it allows to obtain the best of the two worlds: the speed of a low-level compiled language and the flexibility of a high-level interpreted language.

There are many solutions to wrap C++ classes into Python classes; among the most relevant we recall SWIG and SIP. We settled however to use the Boost.Python libraries (see Boost.Python), which deal brilliantly with the heavily-templated code of Piranha and do not require any other interface file to work: a C++ compiler is all that is needed.

7.2 Pyranha: brief overview

We called the set of Piranha's classes and methods which we exposed to Python *Pyranha*. To differentiate between the available manipulator implementations (Fourier series, series with polynomial coefficients, etc.) each manipulator is contained inside its own Python module. A `Core` module contains settings and parameters common to all manipulator implementations, such as the parameters for the output format when saving series to a file or printing them to screen, the floating point representation (decimal or scientific), etc.

Piranha Poisson series exposed in Python are full-fledged classes, and thus interaction with them happens through methods. This leads to a neat grouping (encapsulation) of the manipulation capabilities that make sense on a specific type of series inside instantiations of such classes. For instance, a series with complex coefficients will sport the methods `real()` and `imag()` to retrieve the real and imaginary parts of the series, respectively:

```
real_part = complex.real()
imag_part = complex.imag()
```

(Please note that in Python the type of variables is inferred from the signature of the methods, and hence it is not necessary to specify that `real_part` is a `real_series`, as in C++). The `real_series` type does not feature the `real()` and `imag()` methods, since they do not make sense for series with purely real coefficients. Similarly, a `complex_series` won't feature

¹Languages other than Python were initially considered, but in the end we settled with Python because is the language whose object-oriented model resembles most closely C++'s. An almost 1:1 correspondence between C++ and Python classes can hence be achieved.

7 Pyranha, the Python bindings for Piranha

a cosine method, since trigonometric functions on complex Poisson series are not supported in Piranha (yet).

Operator overloading is supported in Python, so that it will be possible to add two Poisson series objects with the familiar syntax

```
P1+=P2
```

Coherently with the object-oriented paradigm, unary mathematical operations are encapsulated inside the classes, so that, for instance, to calculate the cosine and square root of a series we can write:

```
P1_cos = P1.cosine()  
P1_sqrt = P1.sqrt()
```

Since Python is, like C++, a multi-paradigm language, it is also possible to wrap common mathematical functions in procedural calls, which may be more familiar:

```
P1_cos = cos(P1)  
P1_sqrt = sqrt(P1)
```

The power and cleanness of OO syntax, although, are, in our opinion, compelling reasons to overcome its initial unfamiliarity.

Another interesting possibility when mixing C++ and Python lies in the capability of extending exposed C++ classes with pure Python methods. This way it is possible, for instance, to add a `plot()` method to the series classes that will print a graphical representation of the series using one of the available Python graphical libraries.

7.2.1 An interactive graphical environment

By itself, Pyranha makes available Piranha's capabilities to Python. This means that, for instance, it is possible to write high-level manipulations as Python scripts or functions leveraging the lower-level manipulations provided by the Piranha's engine.

The other interesting application is to use interactively Pyranha from a Python command prompt. The default Python command prompt is usable, but rather spartan. More featureful Python prompts have been developed, and among the many available ones we recommend IPython.

This Python shell provides many interesting features, such as (taken from the project homepage):

- Dynamic object introspection. It is possible to access documentation strings, function definition prototypes, source code, source files and other details of any object accessible to the interpreter with a single keystroke.
- Completion in the local namespace, by typing TAB at the prompt. This works for keywords, methods, variables and files in the current directory.
- Numbered input/output prompts with command history (persistent across sessions and tied to each profile), full searching in this history and caching of all input and output.

- User-extensible “magic” commands. A set of commands prefixed with “%” is available for controlling IPython itself and provides directory control, namespace information and many aliases to common system shell commands.
- Alias facility for defining your own system aliases.
- Complete system shell access. Lines starting with “!” are passed directly to the system shell, and using “!!” captures shell output into python variables for further use.
- Background execution of Python commands in a separate thread. IPython has an internal job manager called jobs, and a convenience backgrounding magic function called “%bg”.
- The ability to expand python variables when calling the system shell. In a shell command, any python variable prefixed with “\$” is expanded. A double “\$\$” allows passing a literal \$ to the shell (for access to shell and environment variables like “\$PATH”).
- Filesystem navigation, via a magic “%cd” command, along with a persistent bookmark system (using “%bookmark”) for fast access to frequently visited directories.
- Automatic indentation (optional) of code as-you-type (through the readline library).
- Macro system for quickly re-executing multiple lines of previous input with a single name.
- Session logging.
- Session restoring: logs can be replayed to restore a previous session to the state in which it was left.
- Verbose and colored exception traceback printouts.
- Auto-parentheses: callable objects can be executed without parentheses: “sin 3” is automatically converted to “sin(3)”.
- Auto-quoting: using ‘, ’ as the first character forces auto-quoting of the rest of the line: ‘my_function a b’ automatically becomes ‘my_function(“a”, “b”)’.
- Extensible input syntax. You can define filters that pre-process user input to simplify input in special situations. This allows, for example, pasting multi-line code fragments which start with “> >” or “...” such as those from other python sessions or the standard Python documentation.
- Flexible configuration system.
- Embeddable. IPython can be called as a python shell inside other python programs. This can be used both for debugging code or for providing interactive abilities.
- Profiler support.

7 Pyranha, the Python bindings for Piranha

One of the advantages of exposing Piranha's capabilities in a language like Python is the availability of a vast set of free libraries which can considerably improve and enhance the usage of the manipulator. We are mainly referring to graphics and GUI (Graphical User Interface) libraries, which can be used to provide a user interface similar (potentially much superior, in our opinion) to those offered by packages like Mathematica and Maple.

In Pyranha we use matplotlib (see matplotlib) to provide plotting support. At the moment 2D plotting is used to visualize Fourier series, to compare them term by term and to display the results of precision tests. We also provide a minimal GUI through the PyQt libraries (see PyQt), which is a set of Python bindings to the cross-platform GUI libraries Qt (see Qt). This interface allows to set some parameters of the manipulator. Figure 7.1 is a screenshot of Pyranha under GNU/Linux in which both the GUI and a graph produced with matplotlib are visible.

Both the GUI and the graphing capabilities are works in progress, and are expected to acquire more features in the near future.

7 Pyranha, the Python bindings for Piranha

8 APPLICATIONS

IN this chapter we are going to present briefly a couple of illustrative examples of computational tasks in the field of Celestial Mechanics that can be efficiently performed with the aid of a specialised manipulator like Piranha.

These examples involve the manipulation of Fourier series, and are related to the research problems that initially spurred our interest towards specialised algebraic manipulators.

8.1 Harmonic development of the TGP

In the Earth-Moon system the lunar tide-generating potential (briefly, TGP) arises from the force experienced by an observer in the non-inertial reference frame linked to the Earth's surface and generated by the Moon's gravitational pull. It differs from the lunar gravitational potential in an inertial reference frame by a corrective term which takes into account the non-inertiality of the terrestrial reference frame, which itself is subject to the lunar gravitational force.

The classical formulation for the TGP in the simplified model in which the Earth and the Moon are considered, dynamically, as point masses and the observer is assumed to lie on the spherical surface defined by the mean radius of the Earth, is the following (see, for instance, Doodson [1922] for a derivation):

$$\mathcal{U}_{\text{TGP}} \propto \frac{1}{d} \sum_{n=2}^{\infty} \left(\frac{a}{d}\right)^n P_n(\cos \psi), \quad (8.1)$$

where d is the Earth-Moon distance, a is Earth's mean radius, P_n are Legendre polynomials and ψ is the angular separation between the vectors connecting the centre of the Earth with the Moon and with the observer.

The expression of the TGP is usually manipulated to separate the so-called astronomical contribution (i.e., the time-varying part of eq. (8.1)) from the geographical contribution (which depends only on the position of the observer on the Earth's surface measured in the terrestrial reference frame, and which is hence constant). The astronomical contribution is classically noted as $B_{nm}(t)$, and expressed as

$$B_{nm}(t) \propto a \left(\frac{a}{d(t)}\right)^{n+1} Y_n^m(\bar{\delta}(t), \alpha(t)) e^{-im\theta_g(t)}, \quad (8.2)$$

where Y_n^m denotes a spherical harmonic (see Appendix A for its definition). The time dependence is expressed by the spherical geographical coordinates of the Moon ($d(t)$, $\delta(t)$, $\alpha(t)$) (distance, equatorial latitude and right ascension) and by the angle $\theta_g(t)$, representing the Greenwich mean sidereal time (which accounts for Earth's rotation).

8 Applications

The astronomical contribution is then usually decomposed harmonically (or *spectrally*) into single components (or *tidal waves*); such decomposition can be used in a variety of fields:

- calculation of the terrestrial nutation motion,
- calculation of the perturbations on the motion of artificial satellites,
- measurements of the terrestrial orientation,
- detection of the alleged oscillations of the terrestrial core.

To be useful in such contexts the harmonical decomposition of the TGP must be expressed in the following form:

$$B_{n,m}(t) = \sum_k C_k \cos(\Theta_k), \quad (8.3)$$

where Θ_k is a linear combination of time-dependent arguments. This form is equivalent to a Fourier series (as defined in §2.1.4).

The goal of the harmonic decomposition of the TGP is hence to obtain eq. (8.3) from eq. (8.2). The decomposition can be carried out analitically using an analytical theory for the motion of the Moon, such as the aforementioned ELP2000 theory (see Chapront-Touzé and Chapront [1988]). This theory expresses as Poisson series the lunar spherical coordinates in an ecliptical reference frame. For instance, for the main problem of the lunar theory, the formulas are:

$$\rho = \sum_{i_1, i_2, i_3, i_4} R_{i_1, i_2, i_3, i_4} \cos(i_1 D + i_2 l' + i_3 l + i_4 F), \quad (8.4)$$

$$\beta = \sum_{i_1, i_2, i_3, i_4} M_{i_1, i_2, i_3, i_4} \sin(i_1 D + i_2 l' + i_3 l + i_4 F), \quad (8.5)$$

$$\lambda = \sum_{i_1, i_2, i_3, i_4} L_{i_1, i_2, i_3, i_4} \sin(i_1 D + i_2 l' + i_3 l + i_4 F) + \omega_1. \quad (8.6)$$

where (ρ, β, λ) are the ecliptical radius, latitude and longitude respectively, (R, M, L) are numerical coefficients and (D, l', l, F, ω_1) are linear combinations of the well-known Delaunay arguments.

These formulas, however, cannot be substituted directly inside eq. (8.2), since in that formula the *equatorial* spherical coordinates appear, not the *ecliptical* ones. In order to achieve the desired harmonical decomposition we will have then to perform the following steps:

1. transform the theory of motion, by applying a rotation, so that it can be used inside eq. (8.2),
2. apply the spherical harmonic to the transformed theory.

The two steps can be combined together by directly rotating the spherical harmonic using Wigner's theorem for the rotation of spherical harmonics (see Appendix A for the formulation of the theorem), hence predisposing the spherical harmonic to accept the ecliptical coordinates instead of the equatorial ones.

All these steps can be performed in Piranha, since they involve:

- elementary arithmetics on Poisson series;
- complex exponentiation of Poisson series;
- spherical harmonics of Poisson series;
- inversion of Poisson series (since in eq. (8.2) the inverse of the lunar distance appears).

Methods to compute all these operations have been described in Chapter 3¹.

This procedure can easily be extended to deal with a more accurate physical model that includes:

- figure perturbations,
- planetary perturbations,
- precise modelling of nutation and precession effects,

all expressed analytically as Poisson series. Results of this work have been presented in Casotto and Biscani [2004a], Casotto and Biscani [2004b] and Casotto and Biscani [2007].

8.2 Perturbations in the Saturn planetary system

It is easily shown (for a derivation see, for instance, Murray and Dermott [2000], Chapter 6) that, in a system of self-gravitating bodies dominated by a massive primary body, the formula for the TGP, eq. (8.1), is analogous to the expression for the perturbing gravitational potential U_P exerted on a test particle by the secondary bodies. In other words:

$$U_P \propto \sum_i \sum_{n=2}^{\infty} \frac{1}{d_i} \left(\frac{r}{d_i} \right)^n P_n(\cos \psi_i), \quad (8.7)$$

where the summation over i is intended over all the secondary bodies, and r is now the distance of the test particle from the primary's centre of mass². If a theory of motion for the system is available, it is then possible to use the procedure described in the previous section to calculate the spectrum of the gravitational perturbation over a spacecraft or a small moon in a planetary system.

The Saturn planetary system is a particularly interesting case, because of its rich dynamical complexity and because of the presence of the Cassini probe since July 2004. The most accurate theory of motion for the Saturnian planetary system is today the TASS theory (*Théorie Analytique des Satellites de Saturne*, Vienne and Duriez [1995]). In the next paragraphs we will show how TASS must be transformed for use in the calculation of the spectrum of the gravitational potential.

¹We note here that it took A.T. Doodson years of work to perform the calculations needed to obtain the first harmonic expansion of the TGP in 1922, and the assistance of tens of human calculators. When we first implemented this methodology, we employed the well-known generic algebraic manipulator Mathematica, which could execute the calculations in around nine hours. With Piranha execution time is in the range of seconds.

²Eq. (8.7) holds when the test particle lies inside the orbits of all secondary bodies. When the particle lies outside, a similar formula holds. We will consider here only the internal case.

8.2.1 Elliptical orbital elements

TASS is expressed in elliptical orbital elements; such formulation is common among theories of motion, since classical orbital elements are undefined under certain conditions (zero inclination, circular orbits, etc.). The elliptical orbital elements are defined as follows:

$$z = ee^{i\varpi}, \quad (8.8)$$

$$\zeta = \sin\left(\frac{i}{2}\right) e^{i\Omega}, \quad (8.9)$$

$$p = \frac{n}{N} - 1, \quad (8.10)$$

$$\lambda = Nt - iq. \quad (8.11)$$

Here e , i , ϖ , Ω and n have the usual meaning of eccentricity, inclination, longitude of pericenter, longitude of ascending node and mean motion respectively. z and ζ are called *Lagrange variables*, and, since they are complex quantities, they encapsulate four classical orbital elements. N is the mean mean motion, so that the linear part of the mean longitude λ is exactly Nt . q is a purely imaginary quantity representing the non-linear part of λ . p represents the deviation of the mean motion n from the mean mean motion N .

By recalling the well-known definitions

$$n = \sqrt{\frac{\mu}{a^3}}, \quad (8.12)$$

$$\lambda = M + \varpi, \quad (8.13)$$

where μ is the gravitational parameter of the system, a is the semi-major axis and M is the mean anomaly, it is then easy to calculate the classical orbital elements from the elliptical ones. Specifically:

$$a = \frac{\sqrt[3]{\mu}}{[N(1+p)]^{\frac{2}{3}}}, \quad (8.14)$$

$$e = |z|, \quad (8.15)$$

$$\varpi = \arg z, \quad (8.16)$$

$$i = 2 \arcsin |\zeta|, \quad (8.17)$$

$$\Omega = \arg \zeta, \quad (8.18)$$

$$M = \lambda - \varpi. \quad (8.19)$$

In TASS the elliptical orbital elements of Saturn's major satellites are given as time-dependent real and complex numerical series which, after some manipulations, can be brought into the form of canonical Poisson series. TASS' reference frame is centered on Saturn's center of mass, and its orientation is defined by the ecliptic plane in the J2000 system. Time is measured in Julian years from J1980.0 (JD=2444240.0).

Mimas' mean longitude λ_1 , for instance, is expressed in radians as

$$\lambda_1 = \underbrace{0.182\,248\,5 + 2\,435.144\,296\,44 \cdot t}_{\lambda_{o1}} + \delta\lambda_1 + \Delta\lambda_1, \quad (8.20)$$

where $\delta\lambda_1$ is a Poisson series representing long period perturbations and $\Delta\lambda_1$ is the sine of another Poisson series representing the short period perturbations. Moreover, λ_{o1} itself is a trigonometric argument of the solutions for the other orbital elements, and it also appears in $\Delta\lambda_1$. The solutions given by TASS, hence, are not canonical Poisson series: they are expressed in the form of trigonometrically nested Poisson series.

We can use Piranha in order to normalize TASS into a purely harmonical form (i.e., a form in which the time dependence is exclusively inside the trigonometric arguments of a Poisson series). We recall indeed that Piranha supports trigonometric operations on Poisson series through the Jacobi-Anger expansion (see §3.2), and hence the substitution of a trigonometric argument for a Poisson series is easily decomposed into products of Poisson series by the elementary trigonometric formulas

$$\cos(\mathbf{a} \pm \mathbf{b}) = \cos \mathbf{a} \cos \mathbf{b} \mp \sin \mathbf{a} \sin \mathbf{b}, \quad (8.21)$$

$$\sin(\mathbf{a} \pm \mathbf{b}) = \sin \mathbf{a} \cos \mathbf{b} \pm \cos \mathbf{a} \sin \mathbf{b}. \quad (8.22)$$

After the normalization into Poisson series, TASS is ready to be transformed into an expression in spherical coordinates, suitable for use in the disturbing potential.

8.2.2 From elliptical orbital elements to radius

As an example of the manipulations involved in the expression of TASS in spherical coordinates we are going to show how to calculate the radius from the elliptical orbital elements. This task involves the transformation of elliptical orbital elements into classical orbital elements, and the resolution of Kepler's equation.

8.2.2.1 Eccentricity e

According to eq. (8.15), e is equivalent to the absolute value of the elliptical orbital element z . The formulation for z in TASS is a complex exponential series:

$$z = \sum_{\mathbf{j}} C_{\mathbf{j}} e^{i(\mathbf{j} \cdot \mathbf{a})}. \quad (8.23)$$

z can hence be seen as a Poisson series with complex coefficients,

$$z = \sum_{\mathbf{j}} [C_{\mathbf{j}} \cos(\mathbf{j} \cdot \mathbf{a}) + iC_{\mathbf{j}} \sin(\mathbf{j} \cdot \mathbf{a})], \quad (8.24)$$

and e can be calculated by means of two multiplications and one square root of Poisson series:

$$e = |z| = \sqrt{\Re[z] \cdot \Re[z] + \Im[z] \cdot \Im[z]}. \quad (8.25)$$

Poisson series representing eccentricities are suitable for the application of the binomial expansion for the calculation of real powers (see §3.1), since they are usually expressed by a constant leading term representing the mean eccentricity and by a much smaller perturbative tail.

8 Applications

8.2.2.2 Complex exponential of M

From eq. (8.8):

$$e^{i\omega} = \frac{z}{e}, \quad (8.26)$$

and from eq. (8.19)

$$e^{iM} = e^{i\lambda} \cdot e^{-i\omega}. \quad (8.27)$$

Hence the complex exponential of M is easily found:

$$e^{iM} = e^{i\lambda} \left(\frac{z}{e} \right)^*. \quad (8.28)$$

The complex exponentiation of λ is performed through the aforementioned Jacobi-Anger expansion, while the inversion of the eccentricity can again be calculated through the binomial expansion.

8.2.2.3 Radius r

r can be found from the eccentric anomaly E as

$$r = a(1 - e \cos E). \quad (8.29)$$

We need to solve Kepler's equation,

$$M = E - e \sin E, \quad (8.30)$$

to calculate the cosine of the eccentric anomaly. We can use the following well-known approximation for E :

$$\begin{aligned} E_0 &= 0, \\ E_1 &= M, \\ &\dots \\ E_{i+1} &= M + \phi_i, \\ &\dots \end{aligned}$$

where we have defined

$$\phi_i = e \sin E_i. \quad (8.31)$$

Then

$$\cos E_{i+1} = \cos(M + \phi_i), \quad (8.32)$$

$$e \sin E_{i+1} = e \sin(M + \phi_i). \quad (8.33)$$

In virtue of the definition of ϕ_i , eq. (8.31), we hence find:

$$\phi_{i+1} = e \sin E_{i+1} = e \sin(M + \phi_i). \quad (8.34)$$

Thus, to calculate $\cos E$ to a certain degree of precision, we must

1. iterate eq. (8.34) to find ϕ to the desired precision,
2. substitute it in eq. (8.32).

Radius r can then be calculated by the relation:

$$r = a(1 - e \cos E). \quad (8.35)$$

The recursive manipulations for the resolution of Kepler's equation involve trigonometric elementary functions, and as such can be handled by Piranha as described earlier.

8.2.2.4 Numerical results and limitations

The application of the method described here for the calculation of the harmonic expansion of the radius of the orbits of Saturn's satellites through TASS leads to good results. We were able, for instance, to produce a Poisson series representing Titan's orbital radius with an accuracy in the order of ~ 10 km with respect to the orbital radius calculated directly with the original TASS series. In Figure 8.1 Titan's orbital radius according to TASS is plotted in blue, while the green line represents the absolute value of the difference with respect to the harmonic expansion of the orbital radius obtained through algebraic manipulation with Piranha.

The results are not as good as they could be because, as it was confirmed by TASS author Prof. Alain Vienne in a private communication, TASS has been produced semi-analytically. This means that many terms in the original series are associated to numerical frequencies and phases that do not correspond to any combination of arguments. Such terms must be discarded because they cannot be handled in an algebraic manner.

These results, despite the limitations introduced by the use of a not completely analytical theory of motion, do show that the methodology described for the transformation of theories of motion is effective. The work towards the harmonic decomposition of the perturbing potential in the Saturn planetary system has been described in Casotto and Biscani [2005].

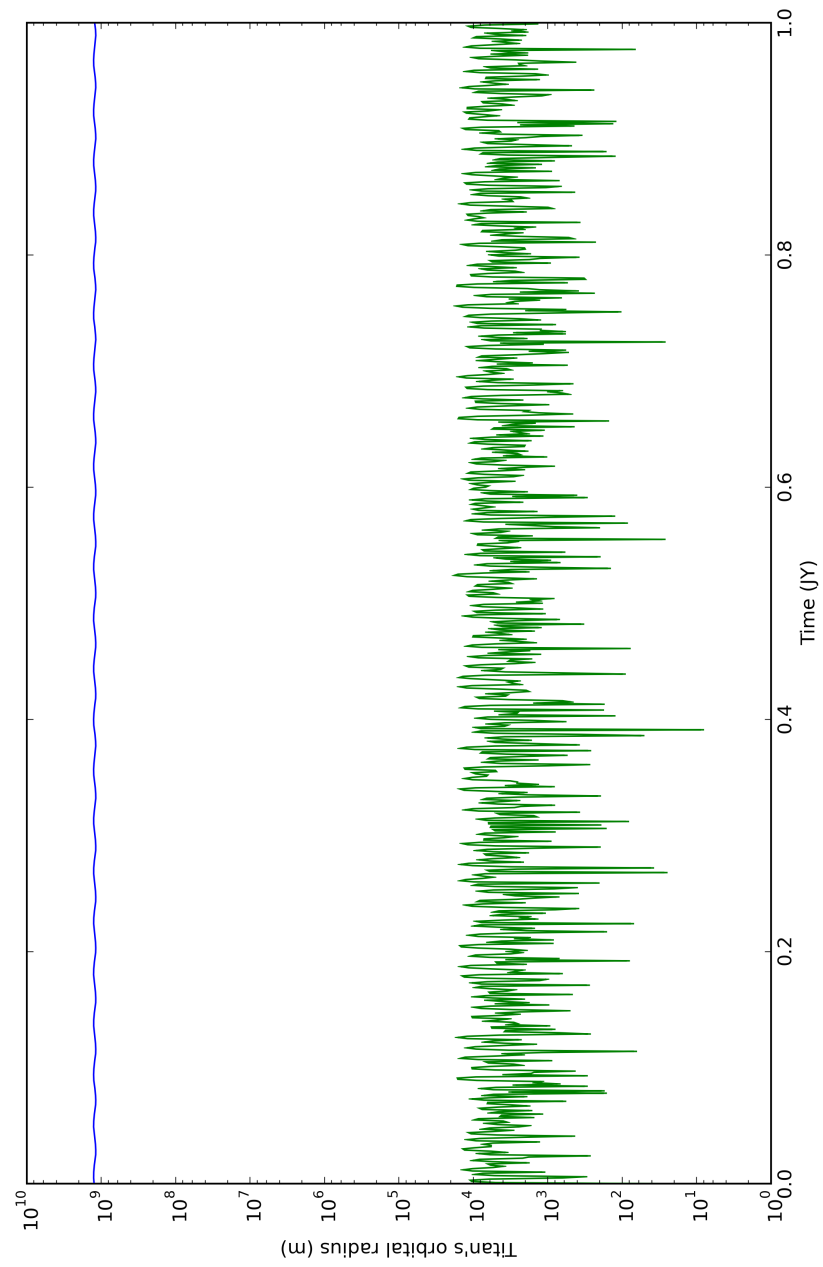


Figure 8.1: Titan's orbital radius from TASS (blue line) is compared to its harmonic development obtained through algebraic manipulations. The absolute value of the difference is plotted in green.

9 FUTURE WORK AND PERFORMANCE REMARKS

THIS chapter presents the future directions for the development of Piranha. Some of the work described here is not “future” any more, having already been implemented while drafting this dissertation.

We also present, as conclusive remarks, some preliminary benchmarks of Piranha against well-known and established algebraic manipulators. Like all benchmarks, the tests reported here must be taken with a grain of salt and by all means they are not intended to be absolute or conclusive. The benchmarks we performed show that Piranha can compete, performance-wise, with the existing specialised algebraic manipulators.

9.1 Generalising coded arithmetics

The technique of coded arithmetics was presented in §6.3 in the context of the manipulation of multivariate sparse polynomials. In §6.4 we suggested a way to extend and generalise this technique, which involves a tailored methodology for the codification of polynomials only during multiplications. The generalisation we present here is also suitable for use in Poisson series.

The following table is a generalisation of the m -variate n -lexicographic representation seen in §6.3:

x_{m-1}	...	x_1	x_0	Code
$e_{m-1,\min}$...	$e_{1,\min}$	$e_{0,\min}$	0
$e_{m-1,\min}$...	$e_{1,\min}$	$1 + e_{0,\min}$	1
...
$e_{m-1,\min}$...	$e_{1,\min}$	$e_{0,\max}$	$e_{0,\max} - e_{0,\min}$
$e_{m-1,\min}$...	$1 + e_{1,\min}$	$e_{0,\min}$	$1 + e_{0,\max} - e_{0,\min}$
$e_{m-1,\min}$...	$1 + e_{1,\min}$	$1 + e_{0,\min}$	$2 + e_{0,\max} - e_{0,\min}$
$e_{m-1,\min}$...	$1 + e_{1,\min}$	$2 + e_{0,\min}$	$3 + e_{0,\max} - e_{0,\min}$
...

The generalisation lies in the fact that each exponent¹ e_k is bounded by two integer values:

$$e_{k,\min} \leq e_k \leq e_{k,\max}, \tag{9.1}$$

¹We call the e_k *exponents*, but they could as well be trigonometric multipliers in a Poisson series.

9 Future work and performance remarks

where

$$k \in [0, m-1] \wedge \{e_{k,\min}, e_{k,\max}\} \in \mathbb{Z}. \quad (9.2)$$

This representation, in other words, differs from the m -variate n -lexicographic one because the exponents

- can have different ranges (i.e., the range is not $[0, n]$ for all the exponents),
- they admit negative values.

If we define

$$\mathbf{e} = (e_0, e_1, \dots, e_{m-1}), \quad (9.3)$$

$$\mathbf{e}_{\min/\max} = (e_{0,\min/\max}, e_{1,\min/\max}, \dots, e_{m-1,\min/\max}), \quad (9.4)$$

$$c_k = 1 + e_{k,\max} - e_{k,\min}, \quad (9.5)$$

$$\mathbf{c} = (1, c_0, c_0 c_1, c_0 c_1 c_2, \dots, \prod_{k=0}^{m-2} c_k), \quad (9.6)$$

$$\chi = \mathbf{c} \cdot \mathbf{e}_{\min}, \quad (9.7)$$

it is fairly easy to show that the lexicographic code of an exponent vector \mathbf{e} in this *generalised lexicographic representation* is

$$\mathbf{h} = \mathbf{c} \cdot \mathbf{e} - \chi. \quad (9.8)$$

Indeed, starting from x_0 's column in the above table, we can intuitively see that each time we move to the next column on the left, such column will feature the same value repeated as many times as needed to cover the whole range of the previous exponent. This consideration leads to the following recursive formula,

$$\begin{aligned} \mathbf{h} &= e_0 - e_{0,\min} + (e_1 - e_{1,\min})(e_{0,\max} - e_{0,\min} + 1) + \\ &\quad (e_2 - e_{2,\min})(e_{1,\max} - e_{1,\min} + 1)(e_{0,\max} - e_{0,\min} + 1) + \dots = \\ &= e_0 + e_1 c_0 + e_2 c_1 c_0 + \dots - e_{0,\min} - e_{1,\min} c_0 - e_{2,\min} c_1 c_0 - \dots = \\ &\quad \mathbf{c} \cdot \mathbf{e} - \mathbf{c} \cdot \mathbf{e}_{\min}, \end{aligned} \quad (9.9)$$

which is analogue to eq. (9.8). It follows then that when adding two exponent vectors \mathbf{e}_1 and \mathbf{e}_2 , provided that the result \mathbf{e}_3 can be represented in the same generalised lexicographic representation, we can write,

$$\mathbf{e}_3 = \mathbf{e}_1 + \mathbf{e}_2, \quad (9.10)$$

and hence, by multiplying both sides by \mathbf{c} and subtracting 2χ ,

$$\mathbf{c} \cdot \mathbf{e}_3 - \chi - \chi = \mathbf{c} \cdot \mathbf{e}_1 - \chi + \mathbf{c} \cdot \mathbf{e}_2 - \chi, \quad (9.11)$$

i.e., by applying eq. (9.8),

$$\mathbf{h}_3 = \mathbf{h}_1 + \mathbf{h}_2 + \chi. \quad (9.12)$$

This is the generalised version of eq. (6.8). In fact, in the case of an m -variate n -lexicographic representation,

$$\mathbf{e}_{\min} = 0, \quad (9.13)$$

$$\mathbf{e}_{k,\max} = n \forall k, \quad (9.14)$$

$$\mathbf{c}_k = (n + 1) \forall k, \quad (9.15)$$

$$\mathbf{c} = \left(1, n + 1, (n + 1)^2, (n + 1)^3, \dots, (n + 1)^{m-1}\right), \quad (9.16)$$

$$\chi = 0, \quad (9.17)$$

and eq. (9.12) simplifies to

$$\mathbf{h}_3 = \mathbf{h}_1 + \mathbf{h}_2. \quad (9.18)$$

These results are in accordance with those described in §6.3.

Since in this representation negative exponents are allowed, it also makes sense to define a similar coded operation for exponents subtraction. The admissibility of subtraction make the generalised lexicographic representation suitable for use also in the operations on trigonometric parts of Poisson series. It is easily proved that, given the relation

$$\mathbf{e}_3 = \mathbf{e}_1 - \mathbf{e}_2, \quad (9.19)$$

the corresponding relation on lexicographic codes holds:

$$\mathbf{h}_3 = \mathbf{h}_1 - \mathbf{h}_2 - \chi. \quad (9.20)$$

For the actual implementation of the schema described above in the manipulation of Poisson series it is better to shift the lexicographic codes up by χ , and hence define:

$$\begin{aligned} \mathbf{h}^{(s)} &= \mathbf{c} \cdot \mathbf{e}, \\ \mathbf{h}_{3,\pm}^{(s)} &= \mathbf{h}_1^{(s)} \pm \mathbf{h}_2^{(s)}. \end{aligned} \quad (9.21)$$

This way the calculation of the *shifted lexicographic code* $\mathbf{h}^{(s)}$ (which can be used as a perfect hash value in a hash table or an index in an array for the accumulation of terms during series multiplication - see §6.3.1) and the addition/subtraction of multiindices, which are the operations performed most frequently during the multiplication of Poisson series, require a minimum number of clock cycles.

Regarding the range of the representation through generalised lexicographic codes, it must be noted that, since negative codes are admitted through eqs. (9.21), it is not possible to rely on unsigned integers. Signed integers must be employed instead, which means that, e.g., on 32-bits machines, codes in the

$$[-2\,147\,483\,648, +2\,147\,483\,647] \quad (9.22)$$

range are admitted.

To establish whether a generalized lexicographic representation can be used during the multiplication of two series or not, it will be enough to analyse the two series' trigonometric multipliers and find their upper and lower limits, which can be used to establish the minimum and maximum

9 Future work and performance remarks

values of the resulting series' multipliers; these minimum and maximum values, in turn, will be used to build the coding vector. This analysis is performed in linear time, and hence it won't weigh significantly during series multiplication (whose complexity is quadratic). The only caveat is to use arbitrary long integers² during the analysis to avoid trespassing the numerical limits of hardware integers. Once the suitability of the representation is established, the coding vector can be downcast to hardware integers.

For the decodification of generalised lexicographic codes, eq. (6.9) is readily generalised into

$$\mathbf{e} = \left(\frac{(h^{(s)} - \chi) \% c_0}{1} + e_{\min}, \frac{(h^{(s)} - \chi) \% (c_0 c_1)}{c_0} + e_{\min}, \dots, \frac{(h^{(s)} - \chi) \% \prod_{l=0}^{m-1} c_l}{\prod_{l=0}^{m-2} c_l} + e_{\min} \right). \quad (9.23)$$

In Piranha coded arithmetics is currently used in a three-way multiplication routine for Poisson series, which will select and call one of these algorithms:

1. standard hash-table multiplication algorithm;
2. coded multiplication using hash tables;
3. coded multiplication using lookup tables.

The first algorithm is the straightforward one, described in §5.6.1: terms are multiplied one by one and accumulated in a hash table. This algorithm is selected when the series to be multiplied feature trigonometric multipliers whose coded representation exceeds the boundaries imposed by the range of the longest hardware integer type available (typically 32 bits or 64 bits wide).

The second algorithm still uses hash tables, but it operates on the coded representations of trigonometric multipliers. It operates faster than the first algorithm because, by working directly on codes instead of multiplier-by-multiplier, the operations of addition/subtraction and localization of the multipliers sets are performed in constant time instead of $O(m)$ (where m is the trigonometric width of the series), and memory usage is lower. This algorithm is selected when the multipliers sets can be encoded into hardware integers but the representation in lookup tables (see next paragraph) is too large to fit into RAM memory.

The third algorithm is the fastest: multipliers are coded, multiplied and stored in a lookup table (which is essentially an array). The codes provide directly the position of the terms in the lookup table, so that the overhead of the management of the hash table present in the second algorithm is avoided. This representation requires a lot of memory, since in the lookup table all the possible codes (even those who are not present in the series, which typically will be the majority) have a unique slot. The multiplication routine falls back to the second algorithm if memory consumption exceeds a user-configurable limit or the lookup table is much too sparse (in this case the overhead of setting up the lookup table dominates over the actual multiplication).

This three-way method is completely transparent to the user, and ensures a higher degree of flexibility with respect to other solutions in which the speed-memory tradeoff is hard-coded (i.e., the user has to select pre-emptively the data structures that will be used during multiplication).

²In Piranha the GMP libraries are used for this purpose.

9.2 Benchmarks

9.2.1 Fourier series

The first benchmark we performed to investigate Piranha's performance is a non-truncated multiplication of Fourier series. The series used in this test is the ELP3 series from the ELP2000 lunar theory (see Chapront-Touzé and Chapront [1988]), which was already introduced in the previous chapters. This series consists of 702 terms, and the trigonometric width is six³.

The test involves the multiplication of ELP3 by itself, performed 20 times to minimize the contribution of I/O and program startup. The resulting Fourier series consists of around 11670 terms⁴.

We have tested Piranha against TRIP (see Gastineau and Laskar [2005]), a mature algebraic manipulator for Celestial Mechanics in use at the IMCCE in Paris. TRIP author, Prof. Mickaël Gastineau, kindly agreed to perform the test himself using two versions of TRIP:

1. version 0.98, the “stable” version at the time of this writing (January 2008),
2. version 0.99, the “development” version, not released yet.

Piranha executed the test in two modes:

1. the “plain” mode, which does not use coded arithmetics,
2. the “coded” mode, which takes advantage of coded arithmetics (the codes for the multiplication performed in this test are suitable for use in a lookup table).

The results of this benchmark are presented in Figure 9.1. We caution that the last column on the right is an (optimistic) estimation of the running time, since all the times are relative to Prof. Gastineau's workstation and Prof. Gastineau has not run the test on Piranha's coded arithmetics yet at the time of this writing.

The results show that Piranha can indeed compete with TRIP in this particular test. As expected, coded arithmetics provides a considerable speed boost with respect to the plain method. If hashed coded arithmetic is forced in place of the lookup table (which is the method automatically selected by the algorithm), Piranha's performance is substantially equal to TRIP's (v0.99).

We would like to stress again that the meaning of this benchmark is limited, and that in our view its importance lies more in the fact that it hints that Piranha is on the right track, performance-wise. In our opinion this is quite comforting, because:

- **Piranha uses a different language and different data structures with respect to TRIP.** These results strongly suggest that C++'s performance is adequate for the task, and that the higher level of abstraction, with respect to languages like C or Fortran, does not necessarily compromise execution speed. Moreover, the data structures employed in Piranha

³The original ELP3 series really has five trigonometric arguments, the sixth one was introduced by us during calculations involving the TGP (see §8.1).

⁴The number of terms is not exactly defined because it depends on the properties of the floating point multiplication unit of the hardware on which the test is run and on the user-configurable threshold that establishes when a floating point value is “zero”.

9 Future work and performance remarks

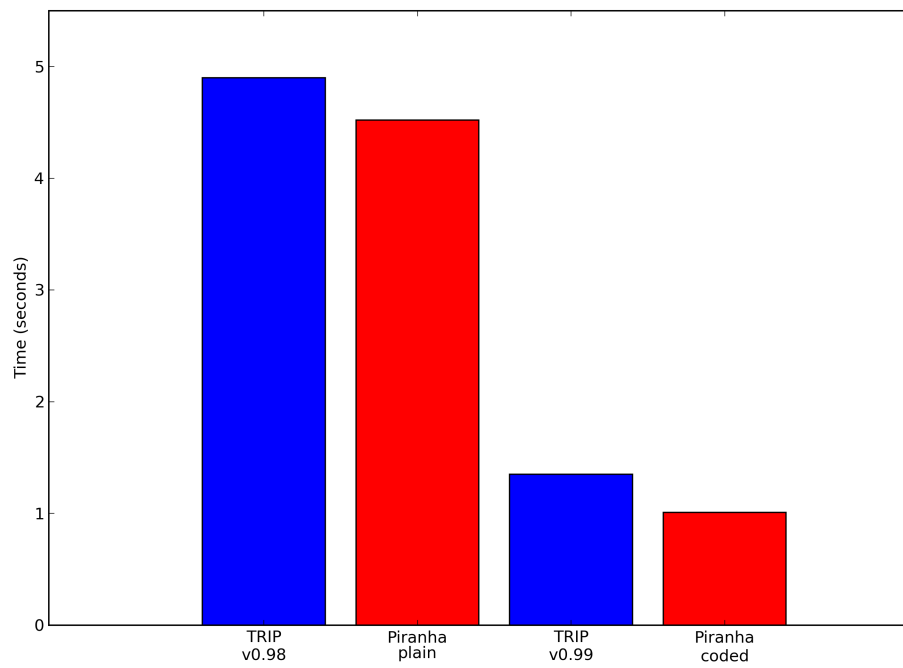


Figure 9.1: Piranha's performance benchmarked against TRIP in a Fourier series multiplication. Piranha's "plain" and "coded" multiplication algorithms are tested with respect to two different versions of TRIP. Running time is expressed in seconds.

seem able to deliver adequate performance, even in the general-purpose implementations provided with standard libraries (we are referring mainly to the hashed data structures, which apparently are not used widely in specialised algebraic manipulators for Celestial Mechanics – we know for sure they are not used in TRIP).

- **Piranha is new and has seen little focus until now on optimization.** The greatest part of the time spent on Piranha has been, until now, focused on the architecture. We hope to be able to increase performance in a variety of ways in the future (see §9.3). By contrast, TRIP has been under active development until the end of the '80s, and has been subject to continuous optimizations.

9.2.2 Multivariate polynomials

The other benchmark we present is performed against PARI/GP (see PARI/GP), a polynomial manipulator intended for number theories computations. The benchmark consists of the following operation:

$$s \cdot (s + 1), \quad (9.24)$$

where

$$s = (1 + x + y + t + u)^{14}. \quad (9.25)$$

The resulting polynomial consists of 35960 terms. Piranha has been benchmarked using coded arithmetics on the exponents and in two modes:

1. “mpz” mode, where the coefficients of the polynomial are arbitrarily-long integers implemented by the GMP libraries (see GMP),
2. “double” mode, in which the coefficients are double-precision floating point values.

For the benchmark we used PARI in the calculator mode (i.e., performing the test from the GP command line). We recall here that PARI uses arbitrarily long numerical values (in this case integers) for the representation of polynomial coefficients.

The results are displayed in Figure 9.2, and show that PARI is ~20% faster than Piranha when operating on arbitrary-size integers. When Piranha uses floating point values the running time, as expected, is considerably shorter. We note that for Celestial Mechanics applications it is often not necessary to retain the absolute numerical precision brought by the use of arbitrary-size types, and that, for the sake of performance, it can be appropriate to employ floating point types (or other hardware numerical types) in such cases.

Regarding this benchmark we also note that:

- at this time coded arithmetics on polynomial does not use lookup tables, only hash tables (performance will considerably improve once lookup tables will be implemented);
- PARI uses a recursive representation for multivariate polynomials, while Piranha uses a “full” representation. This means that the result of eq. (9.25) in Piranha is a sequence of single multivariate monomials, while in PARI it is represented as recursive multiplications of univariate polynomials. If we take the time to spell out all the single monomials in

9 Future work and performance remarks

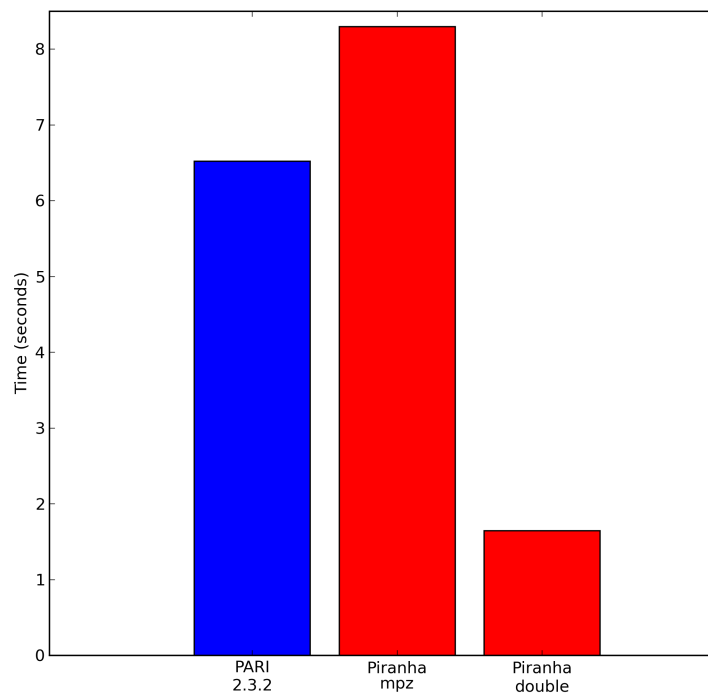


Figure 9.2: Piranha's performance benchmarked against PARI/GP's for a multivariate polynomial multiplication. Piranha is tested using both double precision and arbitrary-size integer coefficients for the polynomials. Running time is expressed in seconds.

PARI (i.e., we explicitly perform all the multiplications of univariate polynomials), PARI's running time is analogue to Piranha's in mpz mode;

- cache memory utilization is not optimized. Once cache-blocking techniques are implemented, performance (especially in mpz mode) is expected to improve.

9.3 Future improvements

Beside the widespread adoption of coded arithmetics, described earlier, we have in mind other improvements for the future developments of Piranha. Some of these improvements are architecture-oriented, others are intended to improve performance and extend features, and yet others aim to offer a better interaction with the manipulator and to increase its usability.

9.3.1 A more generic architecture?

In Chapters 5 and 6 we mentioned how both the Poisson series class and the polynomial class are based on the multiindex container from the Boost libraries. Although the two classes do not share any code at the moment, it is quite evident that many operations performed on Poisson series are very similar to those performed in polynomials (e.g., evaluation, term insertion, etc.): a polynomial can indeed be seen as a Fourier series with different behaviour with respect to the multiplication operation.

One of the tasks we are currently undertaking in Piranha is hence the merging of the concepts of Poisson series and polynomial in a generic *series* structure. This way much code could be shared between the two classes, and the implementation differences could be accounted for with a system of toolboxes similar to the one described in §5.4.

Beyond this, another kind of generalization is sought after. The idea is to have two kind of generic series classes, a *named* series class and an *anonymous* series class. Poisson series, for instance, belong to the first type, since they encapsulate all the information regarding their arguments (as explained in §5.2). Polynomials as stand-alone objects also belong to the first type. However, when we are using polynomials as coefficients in Poisson series, we do not want such polynomial coefficients to know about the arguments, since such information is already included in the Poisson series class. We call this series type anonymous because it has no knowledge about the literal arguments (when such information is needed, e.g., during evaluation, it should be provided by the parent class).

Named and anonymous series could share much of the code, and they would allow to describe a wider range of mathematical entities in a coherent framework. We are referring here mainly to the so-called *echeloned* Poisson series, which are a superset of Poisson series described by the following formula:

$$\sum_{i,j} P_i(x_1, x_2, \dots, x_m) D_j \begin{pmatrix} \cos \\ \sin \end{pmatrix} (i_1 y_1 + i_2 y_2 + \dots + i_n y_n), \quad (9.26)$$

where

$$D_j = \frac{1}{\prod_k (k \cdot n)^{\delta_k}}, \quad (9.27)$$

9 Future work and performance remarks

\mathbf{k} is a vector of integer values, \mathbf{n} is a vector of literal arguments and $\delta_k \geq 0$. Such objects are useful in problems where it is necessary to express the frequencies of trigonometric arguments symbolically, such as in lunar theories (see, for instance, Ivanova [2001] and Rom [1971]).

9.3.2 Improving the implementations of data structures

As we have seen in Chapters 4 and 6, efficient data structures are crucial for high-performance manipulation of Poisson series and polynomials. At the moment in Piranha we are using implementations of data structures already available in standard libraries, which, as we saw in this chapter, have proven to be adequately efficient.

Research on data structures however keeps moving forward, and recently there has been a substantial step forward in the context of hashed data structures. We are referring to the so-called *cuckoo hashing* (see Pagh and Rodler [2001] and Ross [2007]). This hashing technique is noteworthy because it guarantees worst-case $O(1)$ performance (whereas usual hash tables have no such guarantee), and performs very well on modern processor by means of efficient utilization of cache memory (see Zukowski et al. [2006]). Cuckoo hashing is still relatively unknown outside academia, and at this time no implementation exist in standard libraries.

Speaking of cache memory efficient utilization, another possible improvement is the adoption of cache-friendly algorithms, like cache-blocking techniques and partitioning schemes for hash tables. Efficient usage of cache memory is crucial in modern processors, and can lead to very substantial performance boosts.

9.3.3 SIMD instructions and parallelization

As we mentioned in §5.6 Piranha already uses to some extent the SIMD instructions of modern processors (in particular Intel's SSE2 instruction set) and already employs a parallelized algorithm for series evaluation through the use of TBB. In the future we want to substantially increase usage of SIMD instructions and TBB; in particular parallelization and vectorization techniques are appealing for series multiplication. Coded arithmetics is particularly suitable for enhancement with SIMD instructions because of the heavy usage of integer arithmetics.

In the more distant future we would like to investigate the possibility of leveraging cluster environments. The Boost.MPI libraries (see Gregor and Troyer [2008]) are particularly appealing for this task, since they allow to use the well-known MPI libraries for cluster computing in a object-oriented way from C++.

9.3.4 Pyranha improvements

In our opinion a key point for Piranha's future is the enhancement of the Python bindings. Specifically we aim at:

- more graphical and GUI capabilities through tighter interaction with IPython and PyQt,
- exploitation of those capabilities of Python unavailable in C++ (introspection, powerful runtime polymorphism),

- blending of Piranha's data structures with Python native data structures (so that, e.g., series can be accessed like Python dictionaries or lists).

In our intention Pyranha should become the preferred way of interacting with Piranha (although no capability that can be implemented in C++ should be present in Pyranha only).

9.3.5 Interaction with other algebraic manipulators

One of the possibilities we are evaluating for the future of Piranha is to seek integration with SAGE. SAGE (see SAGE) is an umbrella project whose goal is integration and interoperability between algebraic manipulators. SAGE consists of a single frontend which can access a multitude of manipulation engines (including PARI, Singular, Maxima, Magma, etc.). Currently SAGE is lacking a specific Poisson series manipulator, and we think that Piranha could be a good candidate to fill the gap. An interesting possibility brought by the integration with SAGE would be the interoperability with a more general manipulator (like Maxima); this way it would be possible to develop the first steps of an analytical theory of motion with a general manipulator and then switch to Piranha when the Poisson series form is achieved and the need for high-speed computations arises.

We are not seeking integration with commercial packages (like Maple or Mathematica), as seen for instance in Abad and San-Juan [1997]. Piranha indeed is built on top of a stack of entirely Free Software, and it is Free Software itself. We share the ideals of Freedom and transparency expressed by the Free Software movement, and we believe that complete access to the source code with no artificial restrictions is the only possibility in a scientific research context.

9.4 Availability

Although Piranha has been developed in-house until now, the intention from the beginning has been to make the source code available to the public under a Free Software license. Piranha has now a website, which is located at

<http://piranha.tuxfamily.org>.

The source code is in the process of being thoroughly documented with Doxygen, and it will be released as soon as the documentation is complete.

9 Future work and performance remarks

A SPECIAL FUNCTIONS COMMONLY USED IN CELESTIAL MECHANICS

Basic definitions Associated Legendre functions:

$$P_n^m(\mu) = (-1)^m (1 - \mu^2)^{\frac{m}{2}} \cdot \sum_{p=0}^{\lfloor \frac{n-m}{2} \rfloor} (-1)^p \frac{1}{2^p p!} \frac{(2n - 2p - 1)!!}{(n - m - 2p)!} \mu^{n-m-2p}, \quad (\text{A.1})$$

$$P_{nm}(\mu) = (-1)^m P_n^m(\mu). \quad (\text{A.2})$$

Non-normalized spherical harmonics:

$$Y_n^m(\theta, \phi) = P_n^m(\cos \theta) e^{im\phi}. \quad (\text{A.3})$$

Non-normalized solid harmonics:

$$R_n^m(r, \theta, \phi) = r^n Y_n^m(\theta, \phi), \quad (\text{A.4})$$

$$I_n^m(r, \theta, \phi) = r^{-n-1} Y_n^m(\theta, \phi). \quad (\text{A.5})$$

Normalized spherical harmonics:

$${}^n Y_n^m(\theta, \phi) = \sqrt{\frac{2n+1}{4\pi} \frac{(n-m)!}{(n+m)!}} Y_n^m(\theta, \phi) \quad (\text{A.6})$$

$$= (-1)^m \sqrt{\frac{2n+1}{4\pi} \frac{(n-m)!}{(n+m)!}} P_{nm}(\cos \theta) e^{im\phi}. \quad (\text{A.7})$$

Addition Theorems Addition Theorem for non-normalized spherical harmonics:

$$P_n(\cos \psi) = \sum_{m=-n}^n (-1)^m Y_n^m(\theta_1, \phi_1) Y_n^{-m}(\theta_2, \phi_2), \quad (\text{A.8})$$

or, equivalently,

$$P_n(\cos \psi) = \text{Re} \left[\sum_{m=0}^n (-1)^m (2 - \delta_{0m}) Y_n^m(\theta_1, \phi_1) Y_n^{-m}(\theta_2, \phi_2) \right]. \quad (\text{A.9})$$

A Special functions commonly used in Celestial Mechanics

Addition Theorem for normalized spherical harmonics:

$$P_n(\cos \psi) = \frac{4\pi}{2n+1} \sum_{m=-n}^n {}^n Y_n^m(\theta_1, \phi_1) {}^n Y_n^{-m}(\theta_2, \phi_2), \quad (\text{A.10})$$

or, equivalently,

$$P_n(\cos \psi) = \frac{4\pi}{2n+1} \operatorname{Re} \left[\sum_{m=0}^n (2 - \delta_{0m}) {}^n Y_n^m(\theta_1, \phi_1) {}^n Y_n^{-m}(\theta_2, \phi_2) \right]. \quad (\text{A.11})$$

Rotation Theorems Wigner's Rotation Theorem for non-normalized spherical harmonics:

$$Y_n^m(\theta', \phi') = \sum_{k=-n}^n D_{km}^n(\alpha, \beta, \gamma) Y_n^k(\theta, \phi), \quad (\text{A.12})$$

with

$$D_{km}^n(\alpha, \beta, \gamma) = e^{-ik(\alpha - \frac{\pi}{2})} d_{km}^n(\beta) e^{-im(\gamma + \frac{\pi}{2})}, \quad (\text{A.13})$$

$$d_{km}^n(\beta) = \sum_{t=t_1}^{t_2} (-1)^t \frac{(n-k)!(n+m)!}{t!(n+k-t)!(n-m-t)!(m-k+t)!} \cdot \left(\cos \frac{\beta}{2}\right)^{2n-(m-k+2t)} \cdot \left(\sin \frac{\beta}{2}\right)^{m-k+2t}, \quad (\text{A.14})$$

$$t_1 = \max(0, k-m), \quad (\text{A.15})$$

$$t_2 = \min(n-m, n+k). \quad (\text{A.16})$$

Wigner's Rotation Theorem for normalized spherical harmonics:

$${}^n Y_n^m(\theta', \phi') = \sum_{k=-n}^n {}^n D_{km}^n(\alpha, \beta, \gamma) {}^n Y_n^k(\theta, \phi), \quad (\text{A.17})$$

with

$${}^n D_{km}^n(\alpha, \beta, \gamma) = e^{-ik(\alpha - \frac{\pi}{2})} {}^n d_{km}^n(\beta) e^{-im(\gamma + \frac{\pi}{2})}, \quad (\text{A.18})$$

$${}^n d_{km}^n(\beta) = \sum_{t=t_1}^{t_2} (-1)^t \frac{\sqrt{(n+k)!(n-k)!(n+m)!(n-m)!}}{t!(n+k-t)!(n-m-t)!(m-k+t)!} \cdot \left(\cos \frac{\beta}{2}\right)^{2n-(m-k+2t)} \cdot \left(\sin \frac{\beta}{2}\right)^{m-k+2t}, \quad (\text{A.19})$$

$$t_1 = \max(0, k-m), \quad (\text{A.20})$$

$$t_2 = \min(n-m, n+k). \quad (\text{A.21})$$

BIBLIOGRAPHY

- A. Abad and J. F. San-Juan. PSPC: a Poisson Series Processor Coded in C. In K. Kurzynska, F. Barlier, P. K. Seidelmann, and I. Wyrtrzyaszczak, editors, *Dynamics and Astrometry of Natural and Artificial Celestial Bodies*, page 383, 1994.
- Alberto Abad and Felix San-Juan. PSPCLink: a cooperation between general symbolic and Poisson series processors. *Journal of Symbolic Computation*, 24(1):113–122, 1997. ISSN 0747-7171. doi: <http://dx.doi.org/10.1006/jsco.1997.0116>.
- Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth dover printing, tenth gpo printing edition, 1964. ISBN 0-486-61272-4.
- Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70431-5.
- George B. Arfken and Hans J. Weber. *Mathematical Methods for Physicists*. Academic Press, sixth edition, 2005. ISBN 0-120-59876-0.
- Matt Austern. Draft Technical Report on C++ Library Extensions. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf>, 2005.
- I. O. Babaev, V. A. Brumberg, N. N. Vasil’Ev, T. V. Ivanova, V. I. Skripnichenko, and S. V. Tarasevich. UPP - universal system for analytical operations with Poisson series. *Astron. i geod., Tomsk, No. 8, p. 49 - 53*, 8:49–53, 1980.
- Olaf Bachmann and Hans Schönemann. Monomial representations for gröbner bases computations. In *ISSAC ’98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, pages 309–316, New York, NY, USA, 1998. ACM. ISBN 1-58113-002-3. doi: <http://doi.acm.org/10.1145/281508.281657>.
- F. Biscani and S. Casotto. An Advanced Manipulator For Poisson Series With Numerical Coefficients. In *AAS/Division on Dynamical Astronomy 37th Annual Meeting – Halifax (Canada)*, June 2006.
- Francesco Biscani. Sviluppo analitico del potenziale generatore di marea nel sistema Sole-Terra-Luna. Master’s thesis, Università degli studi di Padova – Dipartimento di Astronomia, March 2004.
- Boost. *The Boost C++ Libraries*, 2006. URL <http://www.boost.org>.

Bibliography

- Boost.Python. Seamless interoperability between C++ and the Python programming language. <http://www.boost.org/libs/python/doc/>, 2007.
- S. R. Bourne and J. R. Horton. The design of the Cambridge algebra system. In *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 134–143, New York, NY, USA, 1971. ACM. doi: <http://doi.acm.org/10.1145/800204.806278>.
- P. Bretagnon and G. Francou. Planetary theories in rectangular and spherical variables - VSOP 87 solutions. *Astronomy & Astrophysics*, 202:309–315, August 1988.
- E. Oran Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-307505-2.
- Roger A. Broucke. How to Assemble a Keplerian Processor. *Celestial Mechanics and Dynamical Astronomy*, 2:9, 1970.
- Roger A. Broucke. Construction of rational and negative powers of a formal series. *Commun. ACM*, 14(1):32–35, 1971. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/362452.362478>.
- Roger A. Broucke and K. Garthwaite. A programming system for analytical series expansions on a computer. *Celestial Mechanics and Dynamical Astronomy*, 1(2):271–284, June 1969. ISSN 0923-2958. doi: 10.1007/BF01228844.
- Ernest W. Brown. *Tables of the Motion of the Moon*. Yale University Press, New Haven CT, 1919.
- Ernest W. Brown. *An introductory treatise on the lunar theory*. Dover Publications, 1960.
- J. W. Brown and R. V. Churchill. *Fourier series and boundary value problems*. McGraw-Hill Book Co., Inc., New York, fifth edition, 1993.
- W. S. Brown. A language and system for symbolic algebra on a digital computer. In *SYMSAC '66: Proceedings of the first ACM symposium on Symbolic and algebraic manipulation*, pages 501–540, New York, NY, USA, 1966. ACM. doi: <http://doi.acm.org/10.1145/800005.807955>.
- S. Casotto and F. Biscani. A novel approach to the manipulation of Poisson series in Celestial Mechanics. In *Analytical Methods of Celestial Mechanics – St. Petersburg (Russia)*, July 2007.
- S. Casotto and F. Biscani. The Tidal Potential of the Saturnian Satellite System. In *Celmec IV – A meeting on Celestial Mechanics – San Martino al Cimino, Viterbo (Italy)*, September 2005.
- S. Casotto and F. Biscani. A fully analytical approach to the harmonic development of the tide-generating potential accounting for precession, nutation, and perturbations due to figure and planetary terms. In *Bulletin of the American Astronomical Society*, page 862, May 2004a.
- S. Casotto and F. Biscani. A modern, analytical approach to the harmonic development of the Tide-Generating Potential. In *International Symposium on Earth Tides – Ottawa (Canada)*, August 2004b.

- S. Casotto and F. Biscani. A Poisson series manipulator for application to orbital mechanics. In *Symposium Honoring Byron Tapley's 50 Years of Contributions to Aerospace Education, Research and Service – Austin, TX (USA)*, February 2008.
- J. Chapront and M. Chapront-Touzé. Comparison of ELP-2000 with a numerical integration at the JPL. *Astronomy & Astrophysics*, 103:295–304, November 1981.
- Jean Chapront. Colbert: Manipulateur de séries de Fourier à coefficients littéraux. ftp://synte.obspm.fr/pub/polac/4_programming_tools/2_colbert/colbert.pdf, 2003a.
- Jean Chapront. Gregoire: Manipulateur de séries de Poisson à coefficients numériques. ftp://synte.obspm.fr/pub/polac/4_programming_tools/1_gregoire/gregoire.pdf, 2003b.
- M. Chapront-Touzé and J. Chapront. ELP2000-85: a semianalytical lunar ephemeris adequate for historical times. *Astronomy & Astrophysics*, 190(1-2):342–352, January 1988. ISSN 0004-6361.
- J. R. Cherniak. Techniques for manipulation of long Poisson series. Special Report 328, Smithsonian Astrophysical Observatory, 1970. 12 pp.
- James O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7(2):24–27, 1995. ISSN 1040-6042.
- Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
- J. M. A. Danby, André Deprit, and A. R. M. Rom. The symbolic manipulation of Poisson series. In *SYMSAC '66: Proceedings of the first ACM symposium on Symbolic and algebraic manipulation*, pages 0901–0934, New York, NY, USA, 1966. ACM Press. doi: <http://doi.acm.org/10.1145/800005.807970>.
- R. R. Dasenbrock. A FORTRAN-Based Program for Computerized Algebraic Manipulation. Technical Report 8611, Naval Research Laboratory, 1982.
- James C. Dehnert and Alexander A. Stepanov. Fundamentals of generic programming. In *Selected Papers from the International Seminar on Generic Programming*, pages 1–11, London, UK, 2000. Springer-Verlag. ISBN 3-540-41090-2.
- A. T. Doodson. The harmonic development of the tide generating potential. *Proceedings of the Royal Society*, 100:305–329, 1922.
- Gabriel Dos Reis and Jaakko Järvi. What is generic programming? In *LCSD'05: Library-Centric Software Design – OOPLA Workshop*, 2005.
- Bruce Eckel. *Thinking in C++*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. ISBN 0-13-917709-4.

Bibliography

- I. Z. Emiris and V. Y. Pan. Applications of FFT. In Mikhail J. Atallah and Susan Fox, editors, *Algorithms and Theory of Computation Handbook*, Boca Raton, FL, USA, 1999. CRC Press, Inc. ISBN 0849326494. Produced by Suzanne Lassandro.
- Richard Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bull.*, 37(1):4–15, 2003. ISSN 0163-5824. doi: <http://doi.acm.org/10.1145/844076.844080>.
- Richard Fateman. Can you save time in multiplying polynomials by encoding them as integers? <http://www.cs.berkeley.edu/~fateman/papers/polysbyGMP.pdf>, 2005.
- M. Gastineau and J. Laskar. TRIP 0.98. <http://www.imcce.fr/Equipes/ASD/trip/trip.html>, 2005.
- GMP. GNU Multiple Precision Arithmetic Library. <http://gmp.lib.org/>, 2007.
- Daniel R. Grayson and Michael E. Stillman. Macaulay 2, a software system for research in algebraic geometry. <http://www.math.uiuc.edu/Macaulay2/>, 2007.
- Douglas Gregor and Matthias Troyer. Boost.MPI documentation. <http://www.osl.iu.edu/~dgregor/boost.mpi/doc/>, 2008.
- Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. *SIGPLANNot.*, 41(10):291–310, 2006. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/1167515.1167499>.
- G.M. Greuel, G. Pfister, and H. Schönemann. Singular: a Computer Algebra System for Polynomial Computations. <http://www.singular.uni-kl.de>, 2007.
- Andrew D. Hall Jr. The Altran System for rational function manipulation – a survey. *Commun. ACM*, 14(8):517–521, 1971. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/362637.362644>.
- Scott W. Haney. Beating the abstraction penalty in C++ using expression templates. *Computers in Physics*, 10(6):552–557, Nov/Dec 1996.
- Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002. ISSN 1046-8188. doi: <http://doi.acm.org/10.1145/506309.506312>.
- Jacques Henrard. A survey of Poisson series processors. *Celestial Mechanics and Dynamical Astronomy*, 45(1-3):245–253, March 1988. ISSN 0923-2958.
- Paul Herget and Peter Musen. The calculation of literal expansions. *The Astronomical Journal*, 64:11, 1959. doi: 10.1086/107844.
- W. G. Horner. A New Method of Solving Numerical Equations of all Orders, by Continuous Approximation. *Royal Society of London Proceedings Series I*, 2:117, 1815.

- IPython. *An Enhanced Python Shell*, 2007. URL <http://ipython.scipy.org>.
- T. V. Ivanova. PSP: A new Poisson series processor. In S. Ferraz-Mello, B. Morando, and J. E. Arlot, editors, *IAU Symp. 172: Dynamics, Ephemerides, and Astrometry of the Solar System*, page 283, 1996.
- T. V. Ivanova. A New Echeloned Poisson Series Processor (EPSP). *Celestial Mechanics and Dynamical Astronomy*, 80:167–176, July 2001.
- III W. H. Jefferys. A FORTRAN-based list processor for Poisson series. *Celestial Mechanics and Dynamical Astronomy*, 2:474–480, 1970.
- III W. H. Jefferys. A Precompiler for the Formula Manipulation System Trigman. *Celestial Mechanics and Dynamical Astronomy*, 6:117, 1972.
- Àngel Jorba. A methodology for the numerical computation of normal forms, centre manifolds and first integrals of hamiltonian systems. <http://www.maia.ub.es/~angel/soft.html>, 1998.
- A. Karatsuba and Y. Ofman. Multiplication of many-digit numbers by automatic computers. *Translation in Physics-Doklady*, 7:595–596, 1963.
- Donald E. Knuth. Two notes on notation. *Am. Math. Monthly*, 99(5):403–422, 1992. ISSN 0002-9890. doi: <http://dx.doi.org/10.2307/2325085>.
- Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, second edition, 1998a. ISBN 0-201-89683-4.
- Donald E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, second edition, 1998b. ISBN 0-201-89685-0.
- E. D. Kuznetsov and K. V. Kholshevnikov. Expansion of the Hamiltonian of the Two-Planetary Problem into the Poisson Series in All Elements: Application of the Poisson Series Processor. *Solar System Research*, 38:147–154, March 2004. doi: 10.1023/B:SOLS.0000022825.93837.7d.
- J. Laskar. Manipulation des séries. In D. Benest and C. Froeschle, editors, *Modern Methods in Celestial Mechanics*, page 89, 1990.
- libstdc++ allocators. Documentation for libstdc++'s memory allocators. http://gcc.gnu.org/onlinedocs/libstdc++/20_util/allocator.html, 2007.
- matplotlib. Matplotlib homepage. <http://matplotlib.sourceforge.net/>, 2007.
- Robert T. Moenck. Practical fast polynomial multiplication. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 136–148, New York, NY, USA, 1976. ACM. doi: <http://doi.acm.org/10.1145/800205.806332>.
- M. Moons. Analytical theory of the libration of the moon. *Moon and Planets*, 27:257–284, November 1982.

Bibliography

- Carl D. Murray and Stanley F. Dermott. *Solar System Dynamics*. Cambridge University Press, February 2000.
- David R. Musser and Alexander A. Stepanov. Generic programming. In *ISAAC '88: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, pages 13–25, London, UK, 1989. Springer-Verlag. ISBN 3-540-51084-2.
- Juan Navarro and José Ferrándiz. A new symbolic processor for the earth rotation theory. *Celestial Mechanics and Dynamical Astronomy*, 82:243–263(21), March 2002. doi: doi:10.1023/A:1015059002683.
- OpenMP. Open specifications for Multi Processing, 2007. URL <http://www.openmp.org>.
- A. M. Ostrowski. On two problems in abstract algebra connected with Horner's rule. In *Studies in Math. and Mech. presented to Richard von Mises*, pages 40–48. Academic Press, 1954.
- John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Lecture Notes in Computer Science*, 2161:121, 2001.
- PARI/GP. PARI/GP Development Headquarters. <http://pari.math.u-bordeaux.fr/>, 2007.
- PyQt. PyQt homepage. <http://www.riverbankcomputing.co.uk/pyqt/>, 2007.
- Python. *The Python Programming Language*. The Python Software Foundation, 2007. URL <http://www.python.org>.
- Qt. Qt homepage. <http://trolltech.com/products/qt>, 2007.
- M. V. Ramakrishna and Justin Zobel. Performance in practice of string hashing functions. In *Database Systems for Advanced Applications*, pages 215–224, 1997.
- D. L. Richardson. PARSEC: An Interactive Poisson Series Processor for Personal Computing Systems. *Celestial Mechanics and Dynamical Astronomy*, 45:267, 1989.
- A. Rom. Mechanized Algebraic Operations (MaO). *Celestial Mechanics and Dynamical Astronomy*, 1:301, 1970.
- A. Rom. Echeloned Series Processor (ESP). *Celestial Mechanics*, 3:331–345, September 1971. doi: 10.1007/BF01231805.
- K. A. Ross. Efficient hash probes on modern processors. In *ICDE 2007: Proceedings of the 23rd IEEE International Conference on Data Engineering*, pages 1297–1301, April 2007.

- SAGE. SAGE: Open Source Mathematics Software. <http://www.sagemath.org/>, 2008.
- Félix San-Juan and Alberto Abad. Algebraic and symbolic manipulation of Poisson series. *Journal of Symbolic Computation*, 32(5):565–572, November 2001. ISSN 0747-7171. doi: <http://dx.doi.org/10.1006/jSCO.2000.0396>.
- SIP. A tool for generating Python bindings for C and C++ libraries. <http://www.riverbankcomputing.co.uk/sip/index.php>, 2007.
- Alexander Stepanov. Short history of STL. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-766-X.
- SWIG. Simplified wrapper and interface generator. <http://www.swig.org>, 2007.
- TBB. Intel threading building blocks 2.0 for open source. <http://threadingbuildingblocks.org/>, 2008.
- David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201734842.
- Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. ISSN 1040-6042. Reprinted in *C++ Gems*, ed. Stanley Lippman.
- Alain Vienne and Luc Duriez. TASS1.6: Ephemerides of the major Saturnian satellites. *Astronomy & Astrophysics*, 297:588, May 1995.
- Eric W. Weisstein. Jacobi-anger expansion. <http://mathworld.wolfram.com/Jacobi-AngerExpansion.html>, 2007. From MathWorld – A Wolfram Web Resource.
- Eugene Paul Wigner. *Gruppentheorie und ihre Anwendung auf die Quantenmechanik der Atomspektren*. F. Vieweg & Sohn Akt.-Ges., 1931.
- Marcin Zukowski, Sándor Héman, and Peter Boncz. Architecture-conscious hashing. In *DaMoN '06: Proceedings of the 2nd international workshop on Data management on new hardware*, page 6, New York, NY, USA, 2006. ACM. ISBN 1-59593-466-9. doi: <http://doi.acm.org/10.1145/1140402.1140410>.

Bibliography

NOMENCLATURE

IMCCE Institut de Mécanique Céleste et de Calcul des Ephémérides

SBBST Self-Balancing Binary Search Tree

SIMD Single Instruction, Multiple Data

SSE2 Streaming SIMD Extensions 2

TASS Théorie Analytique des Satellites de Saturne

TBB Intel's threading building blocks library.

TGP Tide-generating potential

VSOP87 Variations Seculaires des Orbites Planetaires

CRTP Curiously Recurring Template Pattern

GCC GNU Compiler Collection

GUI Graphical User Interface

OO Object-Oriented

STL C++ Standard Template Library

TR1 C++ Technical Report 1

Nomenclature

INDEX

- base series class, 43
- Bessel functions of the first kind, 16
- binomial theorem, 13
- Boost libraries, 31
- C++
 - class, 28
 - concept, 43
 - iterator, 31
 - model, 44
 - namespace, 27
- coded arithmetics, 65
- CRTP, 50
- cuckoo hashing, 96
- Delaunay arguments, 10
- dense polynomial, 61
- disturbing function, 9
- echeloned Poisson series, 95
- elliptical orbital elements, 82
- ELP2000, 10
- FFT multiplication, 62
- Fourier series, 8
- garbage collection, 25
- generic programming, 24
- Greenwich mean sidereal time, 79
- hash table, 36
- hash table collisions, 36
- Horner scheme, 62
- inheritance, 29
- integer packing, 54
- Jacobi-Anger developments, 16
- Karatsuba multiplication, 62
- Kepler's equation, 84
- Kronecker's algorithm, 65
- Lagrange variables, 82
- Laurent series, 5
- lexicographic code
 - shifted, 89
- lexicographic ordering
 - m-variate, degree n , 66
 - total degree, 64
- lexicographic representation
 - m-variate, degree n , 66
 - generalized, 88
 - total degree, 64
- load factor, 36
- lookup table, 69
- multiindex container, 38
- operator overloading, 25, 30
- Pochhammer symbol, 14
- Poisson series, 5
 - canonical form, 7
 - length, 6
 - polynomial arguments, 6
 - polynomial width, 6
 - trigonometric arguments, 6
 - trigonometric width, 6
- polynomial exponents, 6
- psymbol class, 47
- Python, 73
- self-balancing binary search tree, 32
- separate chaining, 36
- SIMD, 54

Index

sparse polynomial, 62
SSE2, 54
STL, 25

TASS, 81
TBB, 58
template class, 29
template function, 24
template programming, 24
template specialisation, 30
term packing, 11
theory of motion, 8
Tide-generating potential, 79
toolbox, 48
TR1, 26
tree traversal, 34
trigonometric multipliers, 6
truncation methodologies, 12

VSOP87, 8

Werner's trigonometric formulas, 7